Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediene
Faculté d'Electronique et d'Informatique
Département d'Informatique

Mémoire de Master

Domaine : Mathématiques et Informatique

Filière : Informatique

Spécialité : Systèmes Informatiques Intelligents

Thème

Adaptation de la méta-heuristique BSO aux problèmes d'optimisation continue

Sujet proposé par :
Mme DRIAS Habiba

Présenté par :
Mlle ZOUGGARI Feriel
Mlle BELOUADAH Eden

Devant le jury composé de :

Mr AZZOUNE H.	Président
Mr DAOUDI M.	Membre

Binôme N° : 059/2017
Soutenu le : 21/06/2017

Remerciements

Avant toutes choses, nous remercions Dieu de nous avoir donné la force et le courage de mener à bien ce travail.

* * *

Nous tenons à remercier notre promotrice madame Drias Habiba et l'ensemble des enseignants de nous avoir transmis leur savoir et de nous avoir guidées durant notre cursus.

Nous exprimons nos remerciements aux membres de jury qui nous font l'honneur de juger notre travail.

Nos remerciements vont également à madame Ghislaine de nous avoir aidées dans la correction et la présentation du mémoire.

Enfin, nous sommes reconnaissantes à nos parents pour leur soutien et leurs sacrifices et nous disons merci à nos familles et nos amis pour leurs encouragements.

Sommaire

Table des figures

Liste des tableaux

Introduction générale

Suite au développement des sciences et des technologies, les besoins de l'être humain ont progressé et les problèmes traités sont devenus très complexes et de très grande taille. Ces conditions ont poussé les chercheurs à rechercher des méthodes d'optimisation pour résoudre les problèmes complexes.

Il existe deux catégories de problèmes d'optimisation, les problèmes d'optimisation combinatoire (à variables discrètes) comme ceux du voyageur de commerce, du cube rubique... etc, et les problèmes d'optimisation continue (à variables continues) comme les fonctions d'optimisation globale.

Plusieurs méthodes de résolution ont vu le jour. Dans le cas de l'optimisation combinatoire, on trouve des méthodes exactes et des méthodes approchées. Dans le cas de l'optimisation continue, on trouve des méthodes basées sur la programmation linéaire et d'autres sur la programmation non-linéaire [6].

Nous nous intéressons ici à l'optimisation continue par programmation non-linéaire et plus précisément par les méta-heuristiques qui sont des méthodes globales.

Le but de notre projet est d'adapter la méta-heuristique Bee Swarm Optimization (BSO), qui a été initialement conçue pour les problèmes d'optimisation combinatoire, aux problèmes d'optimisation continue sans perdre l'ossature générale de l'approche initiale.

Le mémoire est construit de cinq chapitres. Dans le premier, nous présentons quelques notions importantes dans le domaine d'intérêt de notre projet. Le deuxième est consacré à l'état de l'art de l'optimisation continue pour étudier l'avancement global des solutions proposées par les chercheurs dans ce domaine. Dans le troisième chapitre, nous présentons CBSO qui est notre propre version continue de BSO. Puis vient l'implémentation de notre proposition et nous finissons par une présentation de nos expérimentations sur des benchmarks publics, avec une analyse des résultats.

Chapitre I

Généralités sur l'optimisation et sur les méta-heuristiques

Introduction

L'Intelligence Artificielle (IA) est un ensemble de techniques informatiques qui permettent à l'ordinateur de résoudre des problèmes complexes nécessitant un raisonnement intelligent comme celui de l'être humain, ou même l'aptitude à acquérir, à apprendre et à utiliser des connaissances. Le terme IA a été introduit la première fois en 1956 lors d'une conférence sur l'intelligence des ordinateurs.

Parmi les domaines les plus importants de l'intelligence artificielle, on trouve la fouille de données (Data Mining), la technologie des agents, la recherche d'information...etc, qui nécessitent des méthodes spécifiques pour résoudre des problèmes d'optimisation complexes.

I.1 Problème d'optimisation

C'est la recherche, parmi un ensemble de solutions, de celle qui maximise ou minimise une fonction objectif mesurant la qualité de cette solution. Cela revient à chercher un optimum global. Un tel problème fait appel à une ou plusieurs techniques d'optimisation.

I.2 L'optimisation

C'est une discipline de la recherche opérationnelle qui a vu le jour au $20^{ème}$ siècle et qui consiste à choisir, parmi plusieurs possibilités, celle qui répond le mieux à un ou plusieurs critères souhaités.

L'outil mathématique intervient dans la résolution des problèmes d'optimisation en les définissant d'une manière formelle et adéquate.

Il existe deux types d'optimisation, l'optimisation combinatoire qui traite les problèmes à variables discrètes, et l'optimisation continue qui traite les problèmes à variables continues.

On distingue dans chacune de ces optimisations deux sous classes :

— l'optimisation mono-objectif qui cherche à optimiser un seul critère,

— l'optimisation multi-objectif qui cherche à optimiser plusieurs critères souvent contradictoires. Par exemple, visiter le plus grand nombre de villes au moindre coût possible. Ce conflit nous pousse à imposer un ordre de préférence entre les critères en définissant une relation d'ordre partiel appelée relation de dominance.

De plus, il existe des problèmes d'optimisation avec et sans contraintes. Une contrainte cherche à restreindre l'espace de recherche et à vérifier l'admissibilité d'une solution sauf qu'elle ne mesure pas la qualité de la solution.

I.2.1 L'optimisation combinatoire

Elle consiste à trouver, dans un ensemble discret, une solution parmi les meilleures réalisables. En général, cet ensemble de solutions est fini mais comporte un très grand nombre d'éléments, et il est décrit de manière implicite, c'est-à-dire par une liste relativement courte de contraintes que doivent satisfaire les solutions réalisables. La notion de meilleure solution est définie par une fonction objectif à maximiser ou à minimiser.

Nous nous intéressons dans ce travail à l'optimisation continue.

I.2.2 L'optimisation continue

Appelée aussi optimisation globale, consiste à trouver les meilleures solutions possibles $x^* \in X$ selon un ensemble de critères $F = \{f_1, ..., f_n\}$. Les critères, appelés fonctions objectifs, sont exprimés sous forme de fonctions mathématiques. Une fonction objectif est une fonction mathématique telle que :

$$f : D \subset \mathbb{R}^n \mapsto \mathbb{R},$$

où D est l'ensemble des points admissibles de l'espace de recherche.

L'optimisation continue fait appel à la notion de continuité d'une fonction mathématique que nous devons expliquer avant de rentrer dans les détails.

- **Continuité d'une fonction en un point[19]**

  Soit α un réel de D.

  — f est continue en α si et seulement si $\lim_{x \to \alpha} f(x) = f(\alpha)$

  — f est continue en α si et seulement si $\lim_{h \to 0} f(a+h) = f(\alpha)$

- **Continuité d'une fonction sur un intervalle[19]**

  — f est continue sur D si et seulement si f est continue en chaque réel α de D.

Dans le cas de l'optimisation d'un seul critère f, un optimum peut être le maximum ou le minimum de la fonction [24].

Les problèmes d'optimisation globale (continue) sont la plupart du temps définis comme des problèmes de minimisation, car maximiser f revient à minimiser $-f$ [9].

La vraie solution optimale pour un problème d'optimisation peut être un ensemble $x^* \in D$ de tous les points optimaux de D, comme elle peut être aussi une seule valeur minimale ou maximale, et cela parce qu'on peut avoir plusieurs ou même une infinité de solutions optimales, cela dépend du domaine qui nous intéresse de l'espace de recherche.

La tâche de n'importe quel bon algorithme d'optimisation globale est de trouver l'optimum global ou au moins des solutions sous-optimales (optimums locaux), c'est-à-dire des optimums globaux pour un sous-domaine du domaine de définition de la fonction[24].

La différence entre un optimum global et un optimum local apparait dans la figure suivante.

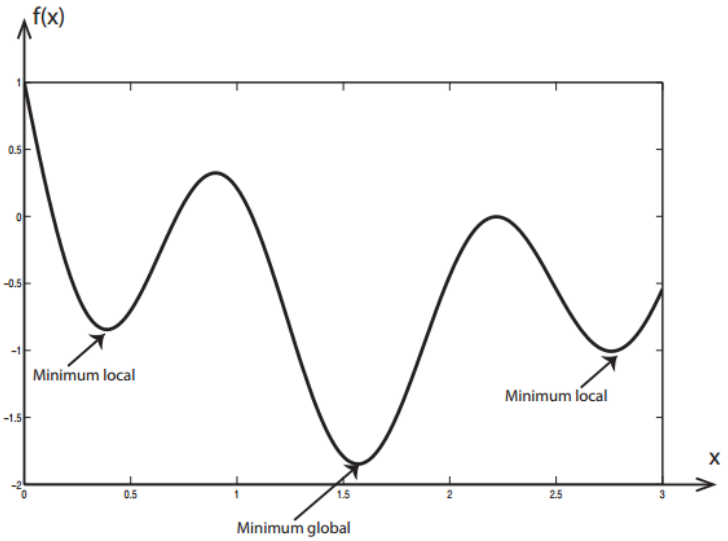


Figure I.1. Différence entre un optimum global et un optimum local [6]

Les fonctions objectifs peuvent être continues, discontinues, linéaires, non-linéaires, convexes, non-convexes, uni-modales, multi-modales, séparables ou non-séparables[24].

— La séparabilité mesure la difficulté d'une fonction. En général, une fonction séparable est plus ou moins facile à minimiser en la comparant avec une autre non-séparable car chaque variable d'une fonction séparable est indépendante des autres. Du coup, la fonction peut être écrite sous forme d'une somme de plusieurs fonctions, chacune à une seule variable[24].

— Le nombre de pics dans le graphe d'une fonction correspond à sa modalité. Une fonction multi-modale est une fonction qui admet plusieurs optimums locaux. Ce type de fonctions est difficile à minimiser car l'algorithme peut stagner dans un optimum local et ne plus en sortir.

— La difficulté d'un problème augmente avec l'augmentation de sa dimension, ce qui représente une barrière pour la plupart des algorithmes d'optimisation.

Les classes des problèmes d'optimisation continue sont variées, d'où la difficulté de trouver une méthode de résolution générale efficiente. C'est la raison pour laquelle nous nous intéressons dans ce travail aux problèmes d'optimisation continue mono-objectif, sans contraintes et de minimisation.

A l'augmentation de la taille du problème, ce dernier devient plus difficile à résoudre et ne peut pas être résolu de manière exacte dans un temps raisonnable, d'où l'intérêt des méthodes approchées qui évitent l'explosion combinatoire par l'exploration partielle de l'espace de recherche mais d'une manière intelligente.

I.3 Les méta-heuristiques

Ce sont des algorithmes génériques qui utilisent des heuristiques pour donner des solutions approchées à des problèmes complexes en utilisant une approche guidée par une technique spécifique.

Les méta-heuristiques se présentent sous deux catégories, celles qui ne sont pas bio-inspirées telles que la recherche Tabou et la recherche dispersée, et celles qui le sont telles que l'algorithme génétique et les algorithmes issus de l'intelligence en essaim (par exemple les algorithmes des colonies de fourmis et d'essaims d'abeilles qui ont été initialement conçus pour des problèmes combinatoires et aussi les algorithmes Bat Algorithm (BA) et Particle Swarm Optimization (PSO) qui ont été développés pour des problèmes continus).

Toutes les méta-heuristiques comportent deux stratégies de recherche :

— l'intensification par recherche locale dans une seule région de l'espace de recherche,

— la diversification qui consiste à changer de région pour en explorer d'autres.

L'importance de la stratégie de recherche varie d'une méta-heuristique à une autre, cela dépend généralement des besoins souhaités :

— plus d'intensification mène à une solution de meilleure qualité,

— plus de diversification mène à trouver une solution en un temps plus réduit,

— un compromis entre les deux mène à une meilleure solution en un temps plus réduit.

I.4 L'intelligence en essaim

C'est une technologie très importante de l'intelligence artificielle qui recouvre un ensemble d'algorithmes à base de population d'agents simples. Un algorithme issu de l'intelligence en essaim se base sur deux principes :

— l'observation à partir des phénomènes naturels intelligents et plus précisément des comportements en groupe,

— la simulation des comportements collectifs des insectes (fourmis et abeilles) et des animaux (poissons et oiseaux) qui traduisent une intelligence collective.

Un essaim peut être vu comme étant un groupe d'agents où chacun fait une tâche élémentaire. L'efficacité de l'essaim peut être remarquée d'un point de vue global qui est établi grâce à l'interaction sociale entre les agents. Cette interaction influence les comportements individuels des agents et les pousse à apporter leur contribution pour atteindre ensemble un but global qui représente le but de l'essaim.

L'intelligence en essaim a été appliquée à plusieurs domaines :

— l'optimisation combinatoire (voyageur de commerce, ordonnancement et planification, transport moderne, télécommunications, etc...),

— l'optimisation continue (optimisation des fonctions numériques),

— les applications de l'intelligence artificielle (traitement automatique du langage, traitement d'image et robotique),

— les applications de divertissement (films et jeux vidéo),

— la fouille de données (Data Mining),

— le routage dans les réseaux,

— les applications Web (recherche, filtrage et sécurité).

Conclusion

Après avoir introduit le domaine de l'intelligence artificielle et plus particulièrement celui de l'optimisation, nous avons abordé la différence entre l'optimisation combinatoire et celle continue. Puis, nous avons présenté la philosophie des méta-heuristiques et de l'intelligence en essaim ainsi que ses différentes adaptations. Le chapitre suivant est consacré aux travaux réalisés par les chercheurs dans le domaine de l'optimisation continue.

Chapitre II

L'optimisation continue

Introduction

Avant de pouvoir proposer n'importe quel algorithme d'optimisation continue, il faudra d'abord établir une étude comparative entre les différentes approches proposées dans le passé. Cela permet d'élargir notre champ de connaissances en tenant compte des points forts des autres algorithmes et de ne pas reprendre les erreurs commises par les autres chercheurs.

Nous ne pouvons pas présenter toutes les approches proposées auparavant, donc nous nous contentons de le faire pour quelques méta-heuristiques et les domaines dans lesquelles elles ont été appliquées pour l'optimisation continue.

II.1 Méta-heuristiques pour les problèmes à variables continues

II.1.1 L'algorithme Particle Swarm Optimization (PSO)

C'est en 1995 que l'algorithme PSO a été proposé par Kennedy et Eberhart pour traiter les problèmes d'optimisation continue [20].

Cette approche simule les comportements des particules telles que les poissons et les oiseaux migrateurs qui se comportent en fonction du comportement global du groupe.

Du point de vu informatique, les particules représentent les solutions candidates d'une population. Les particules évoluent simultanément par partage d'information avec le voisinage. Le rôle de chaque particule est de générer une solution en se basant sur son vecteur vitesse, puis de chercher à améliorer sa position dans l'espace en modifiant son vecteur vitesse à l'instant $t + 1$ grâce à trois critères qui caractérisent son mouvement :

— sa propre vitesse à l'instant t,

— sa meilleure position par le passé,

— la meilleure position connue de la population.

Il y a trois types de comportements de particules :

— égoïste (la particule prend son propre chemin),

— conservatif (la particule conserve sa position),

— panurgique (la particule suit le meilleur comportement du groupe).

L'algorithme PSO est le suivant.

Algorithme II.1 : PSO

Entrées : La condition d'arrêt

Sorties : La meilleure solution trouvée

```
1  début
2      Initialisation aléatoire de l'essaim de particules;
3      tant que la condition d'arrêt est non vérifiée faire
4          gbest ← meilleur pbest_i de toutes les particules;
5          pour chaque particule_i faire
6              v_i^{t+1} ← w v_i^t + r_1 c_1 (x_i − pbest_i) + r_2 c_2 (x_i − gbest);
7              Mise à jour de la position de la particule x_i^{t+1} ← x_i^t + v_i^{t+1};
8          fin pour
9          pour chaque particule_i faire
10             Calculer la valeur de la fonction fitness;
11             si fitness < pbest_i /* meilleure valeur rencontrée pour la
                   particule */
12             alors
13                 pbest_i ← fitness;
14             fin si
15         fin pour
16     fin tq
17 fin
```

PSO a été également développé et appliqué sur un problème de reconfiguration des systèmes de distribution pour minimiser la perte d'énergie. La méthode proposée reformule le problème en tant que problème d'optimisation non linéaire. Elle a été examinée et testée sur les systèmes de test standards IEEE14, IEEE30 et IEEE118 [17].

Encore, PSO a été appliqué sur un problème de modélisation des réseaux de transport pour minimiser le temps de voyage tout en assurant que le budget n'excède pas une certaine limite. PSO a été comparé avec l'algorithme Hybridized Ant Colony Optimization (HACO) qui a été conçu spécialement pour ce genre de problèmes et a donné de bons résultats [1].

Malheureusement, PSO souffre du problème de la convergence prématurée, c'est la raison pour laquelle une version mathématique intégrale modifiée de PSO a été mise au point pour le problème de planification des systèmes de chauffage afin de minimiser le coût du système pour un cycle de vie donné. Il a été prouvé que la version améliorée de PSO (improved PSO, IPSO) résout le problème efficacement et fournit des informations qui montrent qu'il est apte à régler des problèmes de planification réels [23].

II.1.2 L'algorithme Bat Algorithm (BA)

C'est un algorithme bio-inspiré récent, développé par Xin-She Yang en 2010 [32] pour les problèmes d'optimisation globale. Il est inspiré des écholocations naturelles des chauves-souris qui peuvent trouver et discriminer les différents types d'insectes et éviter les obstacles, même dans l'obscurité totale.

Cet algorithme se base sur les trois étapes répétitives suivantes :

— l'évaluation de la performance de chaque chauve-souris,

— la mise à jour des meilleures solutions locales et globales,

— la mise à jour de la position, de la vitesse et de la fréquence de chaque chauve-souris.

L'algorithme BA est le suivant.

Algorithme II.2 : BA

Entrées : Les paramètres de l'algorithme et la condition d'arrêt

Sorties : La meilleure solution trouvée

début
- Définir la fonction objectif $f(x), x = (x_1, ..., x_d)^T$;
- Initialiser la population des chauves-souris $x_i (i = 1, 2, ..., n)$ et la vitesse de chacune des chauves-souris v_i;
- Calculer la fréquence d'impulsion f_i à la position x_i;
- Initialiser les taux d'émissions de pulsation ri et l'intensité A_i;
- **tant que** $t <$ *nombre maximum des itérations* **faire**
  - Générer de nouvelles solutions par l'ajustement des fréquences, et mettre à jour les vitesses et les positions;
  - **si** $rand > r_i$ **alors**
    - Sélectionner une solution parmi les meilleures solutions;
    - Générer une solution locale autour de la meilleure solution sélectionnée;
  - **fin si**
  - Générer une nouvelle solution en volant aléatoirement;
  - **si** $rand < A_i$ *et* $f(x_i) < f(x_*)$ **alors**
    - Accepter les nouvelles solutions;
    - Incrémenter r_i et réduire A_i;
  - **fin si**
  - Classer les chauves-souris et trouver la meilleure solution courante;
- **fin tq**
- Post-traitement des résultats et visualisation;

fin

La mise à jour des vitesses et des positions des chauves-souris est similaire à celle des particules dans l'algorithme PSO. C'est pour cela que BA est considéré comme une extension de PSO avec une recherche locale contrôlée par l'intensité et par le taux d'émission des ondes.

II.2 Méta-heuristiques adaptées aux problèmes à variables continues

II.2.1 L'algorithme Ant Colony Optimization pour les domaines continus ($ACO_{\mathbb{R}}$)

Initialement proposée en 1991 [12], l'approche ACO s'inspire du comportement naturel des fourmis. Ces insectes commencent par une exploration aléatoire de l'espace de recherche, puis déposent de la phéromone en fonction de la quantité et de la qualité de nourriture trouvée. Après une certaine période de temps, la meilleure source de nourriture aura la plus forte concentration de phéromone. Du coup, la colonie de fourmis s'y dirige.

En informatique, l'approche ACO se base sur la construction incrémentale (variable par variable) des solutions. L'ensemble des composants possibles de la solution est défini par le problème.

A chaque étape de construction, une fourmi fait un choix probabiliste pour choisir le prochain composant c_{ij} de la solution à partir de l'ensemble des composants admissibles $N(s^p)$. La règle exacte de ce choix probabiliste varie d'un algorithme à un autre, la meilleure connue est celle de Ant System (AS) [13] :

$$p(c_{ij}|s^p) = \frac{\tau_{ij}^{\alpha} \cdot \eta(c_i j)^{\beta}}{\sum_{c_{il} \in N(s^p)} \tau_{il}^{\alpha} \cdot \eta(c_i j)^{\beta}} \forall c_{ij} \in N(s^P),$$

où :

— t_{ij} est la valeur de phéromone associée au composant c_{ij},

— η est la fonction de pondération qui associe, à chaque étape de construction, une valeur heuristique à chaque composant admissible de la solution $c_{ij} \in N(s^P)$,

— α et β sont des paramètres positifs qui déterminent la relation entre l'information de phéromone et l'information heuristique.

Une extension de l'approche des colonies de fourmis, appelée $ACO_{\mathbb{R}}$, a été présentée pour les domaines continus [22] .

Dans l'algorithme $ACO_{\mathbb{R}}$, Socha et Dorigo proposent d'utiliser une distribution de probabilité continue (qui est une fonction de densité de probabilité, Probability density function en anglais, PDF) au lieu d'utiliser une distribution de probabilité discrète dans le processus de décision.

Dans $ACO_{\mathbb{R}}$, au lieu de choisir un composant $c_{ij} \in N(s^p)$, une fourmi échantillonne une PDF en utilisant un ensemble de solutions appelé archive de solutions. Cette archive

remplace la table de phéromone dans l'approche ACO combinatoire, et la mise à jour de la phéromone se traduit ici par le rajout et le retrait des solutions de cette archive.

Initialement, l'archive de solutions contient k solutions aléatoires (k est un paramètre empirique). Ensuite, à chaque itération, m nouvelles solutions sont générées, évaluées et ajoutées à l'archive qui sera donc de taille $k+m$ (m étant le nombre de fourmis). L'archive est par la suite triée selon les qualités des solutions et seulement les k meilleures sont gardées (la taille de l'archive restera égale à k à la fin de chaque itération).

La représentation de l'archive est la suivante :

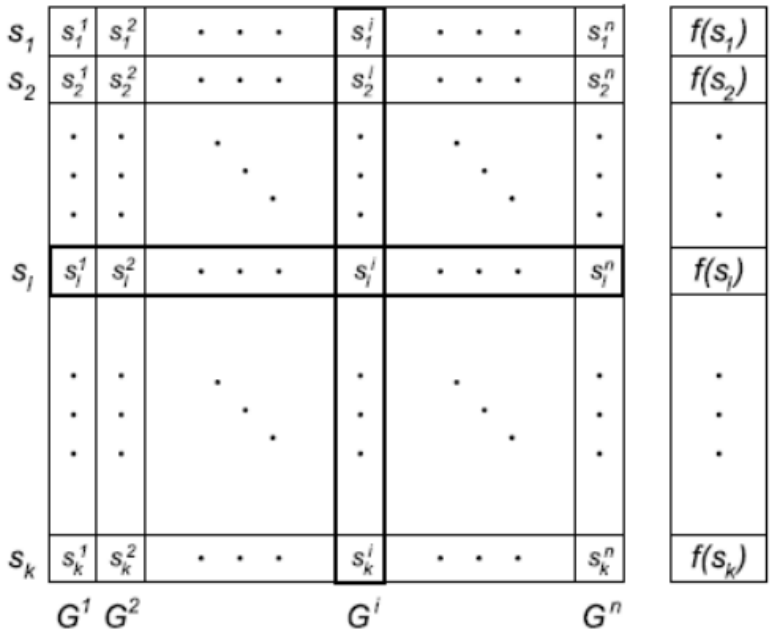


Figure II.1. Archive des solutions dans l'algorithme $ACO_{\mathbb{R}}$

— n est la taille de la solution (dimension de la fonction problème),

— k est le nombre de solutions dans l'archive,

— s_l^i est le $i^{ème}$ composant de la $l^{ème}$ solution.

Chaque ligne de l'archive correspond à une solution construite par une fourmi. La qualité de chaque solution est calculée par la fonction objectif et stockée dans l'archive.

Socha et Dorigo utilisent une fonction de densité de probabilité unidimensionnelle (à une seule variable) et multi-modale basée sur une fonction gaussienne noyau qui représente une somme pondérée de plusieurs fonctions gaussiennes g_l^i où l est l'indice de la solution et i est l'indice du composant. La fonction gaussienne noyau est définie comme suit :

$$G^i(x) = \sum_{l=1}^{k} \omega_l g_l^i(x) = \sum_{l=1}^{k} \omega_l \frac{1}{\sigma_l^i \sqrt{2\pi}} \mathrm{e}^{-\frac{(x-\mu_l^i)^2}{2\sigma_l^{i2}}},$$

où :

— $l \in \{1, ..., k\}$,

— $i \in \{1, ..., n\}$,

— w_l est le poids de la $l^{ème}$ solution.

Le poids de chaque fonction est calculé selon la fonction gaussienne suivante :

$$\omega_l = \frac{1}{qk\sqrt{2\pi}} e^{-\frac{(l-1)^2}{2q^2k^2}}.$$

Cette formule définit le poids comme étant une valeur de la fonction gaussienne avec argument l, moyenne $\mu = 1$ et variance $\sigma = qk$, où q est un paramètre empirique de l'algorithme.

Lorsque q est petit, les meilleures solutions sont fortement préférées et lorsqu'il est grand, la probabilité devient plus uniforme.

En pratique, dans le processus de construction de solution, chaque composant de la nouvelle solution est traité indépendamment des autres. Chaque fourmi choisit une des solutions de l'archive selon son poids. La probabilité p_l de choisir la $l^{ème}$ solution est donnée par :

$$p_l = \frac{\omega_l}{\sum_{r=1}^{k} \omega_r}.$$

Ensuite, à l'étape i, l'algorithme échantillonne la fonction gaussienne associée au $i^{ème}$ composant de la solution choisie en utilisant une fonction de densité de probabilité gaussienne, avec $\mu_l^i = s_l^i$ et σ_l^i donnée par :

$$\sigma_l^i = \xi \sum_{e=1}^{k} \frac{|S_e^i - S_l^i|}{k-1}.$$

Cette formule représente la distance moyenne entre la $i^{ème}$ variable de la $l^{ème}$ solution (solution choisie) et la $i^{ème}$ variable des autres solutions de l'archive, multipliée par un paramètre ξ.

Ce paramètre a un effet similaire à celui du taux d'évaporation de la phéromone dans l'approche ACO. Plus ξ est grand, plus la vitesse de convergence de l'algorithme est petite.

Après le calcul de μ_l^i et σ_l^i, la $i^{ème}$ variable de la nouvelle solution aura comme valeur un nombre aléatoire généré en obéissant à la loi gaussienne $N(\mu_l^i, \sigma_l^i)$.

Le processus de construction de solution est répété m fois pour chaque dimension $i = 1, ..., n$ du problème. Après chaque construction de solution, la mise à jour de la phéromone est traduite par le fait d'ajouter m nouvelles solutions générées à l'archive T et d'éliminer le même nombre de plus mauvaises solutions ; on garde donc le même nombre k

de solutions dans l'archive. Ce processus permet de ne garder que les k meilleures solutions, ce qui va permettre de bien guider les fourmis dans leur processus de recherche.

L'algorithme $ACO_{\mathbb{R}}$ permet d'ajouter des activités qu'une simple fourmi ne peut pas faire : recherche locale sur les solutions construites, récolte d'une information globale pour décider s'il est utile de déposer encore de la phéromone, etc... Cependant, ces activités n'ont pas été appliquées par Socha et Dorigo.

Le pseudo code de l'algorithme $ACO_{\mathbb{R}}$ est donné comme suit :

Algorithme II.3 : $ACO_{\mathbb{R}}$

Entrées : k,m,n,q,ξ et la condition d'arrêt
Sorties : La meilleure solution trouvée

```
1  début
2      Initialiser et évaluer k solutions;
       // Trier les solutions et les mettre dans l'archive
3      T = Trier(S_1...S_k);
4      tant que la condition d'arrêt est non vérifiée faire
           // Générer m nouvelles solutions
5          pour l = 1 à m faire
               // Construire une solution
6              pour i = 1 à n faire
7                  Choisir une fonction gaussienne g_j^i selon les poids;
8                  Échantillonner la fonction g_j^i avec le paramètre μ_j^i · σ_j^i;
9              fin pour
10             Sauvegarder et évaluer la solution générée;
11         fin pour
           // Trier les solutions et choisir les k meilleures
12         T = Meilleure(Trier(S_1...S_{k+m}), k);
13     fin tq
14 fin
```

II.2.2 L'algorithme $ACO_{\mathbb{R}}$ avec la théorie des perspectives ($ACO_{\mathbb{R}}$-PT)

Riadi Indra [27] a proposé une amélioration de l'algorithme $ACO_{\mathbb{R}}$ qui garde le même principe que celui de $ACO_{\mathbb{R}}$ sauf qu'elle modifie la transition d'état lors de la phase de construction. Une solution n'est pas choisie en se basant sur son poids seulement, mais aussi sur son évaluation. C'est pourquoi Indra commence par calculer un point de référence qui est la moyenne des évaluations des solutions de l'archive.

$$point\ de\ référence = moyenne(f(s_1), f(s_2), ..., f(s_k)).$$

Ensuite, il commence à séparer les solutions de l'archive en deux groupes. Les solutions qui ont une évaluation plus petite que celle du point de référence auront un gain négatif par rapport à l'écart du point de référence et les solutions qui ont une évaluation plus grande ou égale à celle du point de référence auront un gain positif.

De plus, une valeur probabiliste est calculée pour chaque solution. Le nouveau poids (perspective) de la solution est obtenu par la multiplication de son gain par sa valeur probabiliste.

La meilleure solution est celle qui a le poids le plus élevé et qui va donc être utilisée pour calculer la nouvelle valeur de phéromone.

II.2.3 L'algorithme Continuous Ant Colony Optimization (CACO)

Cet algorithme, développé par Bilchev et Parmee [4], représente le premier essai pour adapter un algorithme inspiré du comportement des fourmis aux problèmes d'optimisation continue. Dans CACO, les fourmis commencent à partir d'un point appelé *nid*, situé quelque part dans l'espace de recherche. Les bonnes solutions trouvées sont stockées en tant qu'ensemble de vecteurs et proviennent du nid. A chaque itération, les fourmis font un choix probabiliste sur un des vecteurs. Ensuite, elles continuent la recherche à partir du vecteur choisi en faisant des mouvements aléatoires. Les vecteurs sont mis à jour par les meilleurs résultats trouvés.

Bien que les auteurs de CACO prétendent qu'ils se sont inspirés de l'approche ACO originale, CACO présente plusieurs différences avec ACO. En effet, la notion de *nid* n'existe pas dans l'approche ACO. De plus, CACO ne fait pas une construction incrémentale des solutions alors que celle-ci est l'une des caractéristiques principales de ACO. Donc, CACO n'est pas une extension de ACO.

II.2.4 L'algorithme Continuous Interacting Ant Colony (CIAC)

Cette approche, basée sur le comportement des fourmis, a été présentée par Dréo et Siarry [14]. CIAC utilise deux types d'interaction entre les fourmis, l'information stigmergique (points de phéromone déposés dans l'espace de recherche) et la communication directe entre les fourmis. Ces dernières font des mouvements stratégiques en suivant la phéromone et en communicant entre elles. De même que CACO, CIAC n'est pas une extension de ACO car il ne construit pas les solutions de façon incrémentale et est basé sur une communication directe entre les fourmis.

II.2.5 L'algorithme Taboo Search (TS)

La première adaptation de la recherche Tabou aux domaines continus est présentée par Cvijovic et Klinowski en 1995 [9]. Dans cet algorithme, les auteurs ont introduit une structure de voisinage pour l'espace continu qu'ils ont appelée *voisinage conditionnel*. L'espace de solutions S (qui représente un hyper-cube dans $\mathbb{R}^n$, où n est le nombre de variables) est partitionné en des cellules disjointes en divisant en $p_1, p_2, ..., p_n$ partitions, les intervalles de coordonnées au long des axes $x_1, x_2, ..., x_n$.

A chaque itération, n_s points sont tirés à partir de n_c partitions choisies au hasard en utilisant une distribution uniforme. Ces points vont constituer les voisins de la solution courante s^*. La taille du voisinage $N(s^*)$ est égale à $n_c n_s$.

II.2.6 L'algorithme Continuous Tabu Search (CTS)

Une autre extension de la recherche Tabou proposée par Siarry et Berthiau [29] explore la notion de voisinage inventée par N. Hu [25] qui se base sur le principe de disques centrés par la solution.

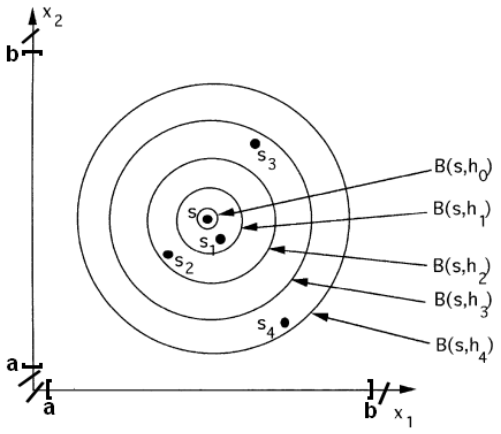


Figure II.2. Partitionnement du voisinage d'une solution dans CTS

Les voisins de la solution sont pris chacun d'une couronne qui sépare les frontières de deux disques consécutifs. Une explication détaillée de cette notion sera expliquée dans le prochain chapitre.

CTS commence à partir d'une solution aléatoire, puis génère un certain nombre fixe de voisins et continue avec le meilleur voisin, même si son évaluation est pire que celle de la solution courante (voisinage en anneau).

Pour éviter de boucler sur les mêmes voisins, chaque voisin généré est inséré dans une liste Tabou circulaire qui garde trace des dernières solutions rencontrées vers lesquelles nous ne devons pas revenir. Cependant, ce processus risque de contourner des mouvements

utiles, ce qui nécessite l'incorporation d'un mécanisme d'aspiration qui permet d'éviter une telle situation.

L'algorithme s'arrête lorsqu'il trouve l'optimum ou lorsqu'il n'arrive pas à améliorer la solution au bout d'un certain nombre d'itérations.

II.2.7 L'algorithme Enhanced Continuous Tabu Search (ECTS)

C'est une adaptation de la recherche Tabou aux domaines continus qui consiste d'abord à faire une diversification pour détecter les régions prometteuses, puis à effectuer une intensification afin de trouver l'optimum dans la région la plus prometteuse [8] .

Le voisinage est défini de la même manière que dans CTS sauf que les auteurs ici considèrent des hyper-rectangles au lieu des disques.

L'algorithme ECTS est le suivant :

Algorithme II.4 : ECTS

Entrées : Les paramètres de l'algorithme et la condition d'arrêt

Sorties : La meilleure solution trouvée

```
1 début
2     Diversification;
3     Chercher la région la plus prometteuse;
4     tant que la condition d'arrêt est non vérifiée faire
5         Intensification;
6         Mise à jour de la meilleure solution;
7     fin tq
8 fin
```

II.2.8 L'algorithme Continuous Reactive Tabu Search (CRTS)

Une amélioration de la recherche Tabou pour les domaines continus a été proposée par Battiti et Tecchiolli [2], et qui s'inspire de l'approche combinatoire Reactive Tabu Search (RTS) en la combinant avec un minimisateur stochastique. Le composant combinatoire sert à localiser les régions prometteuses (qu'ils ont appelées boîtes) à partir desquelles le minimisateur effectue sa recherche locale. La taille des boîtes ainsi que les paramètres de recherche sont adaptés automatiquement à la structure locale de la fonction objectif.

Les auteurs ont développé une fonction d'évaluation des boîtes qui permet d'indiquer si la boîte contient de bonnes solutions. Ils ont testé deux méthodes de calcul et ont exprimé chacune dans un algorithme :

— CRTS$_{average}$ (CRTS$_{ave}$) qui considère la moyenne des évaluations des solutions dans la boîte :

$$X_i : f(B) \equiv \left(\frac{1}{N_B}\right) \sum_{i=1}^{N_B} f(X_i),$$

où N_B est le nombre de points.

— CRTS$_{minimum}$ (CRTS$_{min}$) qui considère le minimum des évaluations des solutions dans la boîte :

$$f(B) \equiv min_{i \in (1,\ldots,N_B)} f(X_i).$$

II.2.9 L'algorithme Continuous Genetic Algorithm (CGA)

L'algorithme génétique se base sur les opérateurs suivants :

— la sélection des individus qui doivent se reproduire par la technique de tirage à roulette ou par tournoi ou par rang,

— le croisement des individus sélectionnés,

— la mutation (modification aléatoire) des nouveaux individus,

— le remplacement des parents par les nouveaux individus.

La nouvelle population sera construite des meilleures individus de la population courante.

La version continue de l'algorithme génétique redéfinit les opérateurs de croisement et de mutation en tenant compte de la taille et de la distribution de la population dans l'espace de recherche [7].

La taille de la population est initialement large pour une meilleure convergence de l'algorithme, mais elle sera réduite au fur et à mesure pour éviter une explosion en termes de temps d'exécution. De plus, pour une meilleure exploration de l'espace de recherche, la population initiale est construite par des solutions suffisamment éloignées les unes des autres et une solution n'est choisie que lorsqu'elle n'appartient à aucun voisinage des autres solutions de la population.

Les auteurs, comme N. Hu[25], utilisent le principe de disque pour explorer le voisinage d'une solution.

L'algorithme CGA est le suivant :

Algorithme II.5 : CGA

Entrées : La condition de convergence

Sorties : La meilleure solution trouvée

```
1 début
2     Définir la fonction coût, le coût et les variables;
3     Choisir les paramètres de l'algorithme;
4     tant que la condition de convergence est non vérifiée faire
5         Choisir les parents;
6         Croisement;
7         Mutation;
8     fin tq
9 fin
```

II.2.10 L'algorithme Enhanced Simulated Annealing (ESA)

Cet algorithme développé par Siarry et ses camarades [30], manipule les problèmes de grande taille en faisant une discrétisation des variables. La recherche aléatoire itérative est remplacée ici par une exploration de plusieurs espaces euclidiens.

Cet algorithme a été prouvé efficace pour résoudre les problèmes de taille 2 à 200 variables.

II.2.11 L'algorithme Evolutionary Strategies (ES)

Différentes adaptations de l'algorithme évolutionniste ES aux domaines continus ont vu le jour à travers les années.

— (1+1)ES [21] se base sur un seul parent qui génère une progéniture par itération. Seul l'individu représentant la meilleure solution est gardé.

— Cumulative Step Size Adaptation (CSA-ES) [18] contrôle la taille du pas global en utilisant un chemin traversé par la population parente à travers un certain nombre de générations,

— Covariance Matrix Adaptation (CMA-ES) [18] est une variante de CSA-ES avec une adaptation aléatoire de la matrice de covariances.

II.2.12 L'algorithme Iterated Estimation of Distribution Algorithm (IDEA)

C'est une adaptation de Estimation of Distribution Algorithms (EDAs) pour les problèmes d'optimisation continue, développée par Bosman et Thierens en 2002 [5]. Pour estimer la distribution de la population parente, IDEA exploite le fait que chaque probabilité jointe multi-variée peut être exprimée en tant que factorisation conditionnelle :

$$P(x_i, ..., x_n) = \prod_{i=1}^{n} P(x_i | x_{i+1}, x_{i+2}, ..., x_n).$$

Le modèle probabiliste de la population parente est reconstruit à chaque itération.

II.2.13 L'algorithme Mixed Bayesian Optimization Algorithm (MBOA)

Cet algorithme développé par Ocenasek et Schwarz [26], est basé sur un réseau bayésien avec des structures locales sous forme d'arbres de décision qui capturent les dépendances mutuelles entre les individus parents. MBOA est une extension continue de l'algorithme hierarchical BOA initialement conçu pour le domaine binaire. MBOA peut traiter les variables de type mixte (discret et continu).

II.2.14 L'algorithme INTEROPT

C'est une méthode numérique conçue par Bilbro Griff et Snyder Wesley [3], pour optimiser les fonctions non-convexes. Elle est basée principalement sur l'approche du recuit simulé mais traite les problèmes à variables continues. La méthode est complètement générale et capable de résoudre les problèmes qui ont une grande taille.

Conclusion

Chaque méta-heuristique a ses avantages et ses inconvénients. La stratégie de recherche diffère d'un algorithme à un autre mais le but est toujours le même, c'est d'essayer d'arriver à la meilleure solution au problème dans les meilleurs délais possibles.

Nous avons présenté deux catégories de méta-heuristiques, celles conçues pour des problèmes à variables continues et celles développées à l'origine pour des problèmes à variables discrètes et qui ont été transformées pour s'adapter aux problèmes à variables continues. Le chapitre suivant est consacré à l'algorithme Bee Swarm Optimization (BSO) et à notre propre adaptation de ce dernier aux problèmes d'optimisation continue.

Chapitre III

Adaptation de BSO aux problèmes d'optimisation continue

Introduction

L'objectif de notre travail est d'adapter la méta-heuristique BSO aux problèmes d'optimisation continue sans trop affecter sa structure initiale. Après un rappel du principe général de BSO, nous passons en revue les domaines dans lesquels elle a été appliquée. Puis, nous exposons notre contribution qui consiste à son adaptation à l'optimisation continue.

III.1 L'algorithme Bee Swarm Optimization (BSO)

III.1.1 Principe

L'algorithme BSO classique s'inspire du comportement naturel des essaims d'abeilles.

L'abeille éclaireuse quitte la ruche à la recherche du nectar. Quand elle le trouve, elle revient à la ruche avec un peu de récolte, puis elle fait une danse circulaire. Pour les sources de nourriture qui sont loin de la ruche, l'abeille éclaireuse change petit à petit sa danse en une danse en huit (la danse d'abeille est riche en informations parce qu'elle indique la quantité de la nourriture trouvée, la distance qui la sépare de la ruche et la direction de sa source).

L'essaim d'abeilles se dirige vers la source de nourriture la plus riche, même si elle est la plus éloignée (ce comportement a été confirmé par l'expérience de Seeley et ses camarades [28]).

Du point de vue informatique, on associe à l'abeille éclaireuse un sommet de référence qui est une solution construite aléatoirement ou en suivant une heuristique. A partir de

ce sommet, on génère n autres solutions en utilisant le paramètre $Flip$ permettant de déterminer un ensemble de points de l'espace de recherche à explorer. L'ensemble de ces solutions définit la zone de recherche.

Ensuite, on attribue chaque point ou solution de la zone de recherche à une abeille, qui fera une recherche locale afin d'aboutir à un optimum local. L'ensemble des optimums locaux est mis dans une table appelée $Danse$. Le meilleur optimum local à une itération sera considéré comme le nouveau sommet de référence de la prochaine itération.

Pour éviter de boucler indéfiniment dans la même zone de l'espace de recherche, le sommet de référence est sauvegardé à chaque fois dans une liste appelée $Tabou$.

Dans un premier temps, le choix du sommet de référence est fait en se basant sur les qualités des solutions (phase d'intensification). Puis, si après une certaine période de temps, on n'arrive pas à améliorer la solution, on utilise le critère de diversité pour changer de région et en explorer d'autres (phase de diversification).

L'algorithme BSO est le suivant :

Algorithme III.1 : BSO

Entrées : Les paramètres de l'algorithme et la condition d'arrêt
Sorties : La meilleure solution trouvée

```
1  début
2      Soit Sref la solution trouvée par l'abeille éclaireuse;
3      tant que la condition d'arrêt est non vérifiée faire
4          Insérer Sref dans la liste Tabou;
5          Déterminer la zone de recherche à partir de Sref;
6          Attribuer une solution s de la zone de recherche à chaque abeille;
7          pour chaque abeille faire
8              Recherche locale avec la solution s;
9              Mettre l'optimum local dans la table Danse;
10         fin pour
11         Choisir le nouveau sommet de référence Sref;
12     fin tq
13 fin
```

III.1.2 Applications

C'est en 2005 que l'algorithme BSO a été proposé et appliqué sur le problème de satisfiabilité MAXW-SAT puis adapté à plusieurs domaines d'application parmi lesquelles la fouille de données et la recherche d'information [16].

Une méthode hybride a été proposée pour prendre en charge les instances volumineuses du problème SAT. Cette méthode se base essentiellement sur la méta-heuristique bio-

inspirée BSO et sur le paradigme multi-niveaux. Dans ce travail, on perçoit l'importance de la combinaison de deux approches prometteuses afin de bénéficier de leurs avantages [10].

Une combinaison entre le Clustering (une tâche du Data Mining) et la méta-heuristique BSO a été mise au point pour améliorer l'efficacité de la résolution du problème SAT. L'approche suivie consiste à explorer judicieusement l'espace de recherche avant de lancer le processus de recherche de solutions qui va réduire la complexité des instances volumineuses du problème SAT. Deux méthodes ont été proposées. La première consiste à intégrer le Clustering dans la structure originale de BSO, qui va mener à suggérer une version avancée de BSO. La deuxième méthode consiste à faire un Clustering sur les données avant de lancer BSO. Cette manière de procéder aide à réduire le nombre de clauses et le nombre de variables des instances SAT et donc le temps d'exécution [15].

D'autres auteurs ont proposé un nouvel algorithme de recherche de règles d'association basé sur une version améliorée de BSO avec trois heuristiques différentes pour explorer l'espace de recherche. Cette approche a été implémentée et testée sur différentes bases de données de petite, moyenne et grande taille. Les résultats obtenus montrent que l'approche proposée est meilleure que certains algorithmes de la littérature en termes de qualité et de temps d'exécution [11].

Nous nous intéressons dans ce travail à l'adaptation de BSO aux problèmes d'optimisation continue afin d'élargir son champ d'application. Nous allons présenter notre contribution à cette problématique en développant une version de BSO pour les problèmes à variables continues, que nous avons appelée Continuous Bee Swarm Optimization (CBSO).

III.2 Continuous Bee Swarm Optimization (CBSO)

Le problème que nous traitons consiste en des fonctions mathématiques à plusieurs dimensions (variables). A noter que la fonction problème est elle-même la fonction objectif qu'on cherche à minimiser. Le but est donc de trouver les valeurs réelles que prennent les variables ($x_1, x_2, ..., x_n$ où n est la dimension du problème) pour lesquelles la fonction donne l'image la plus petite possible (les bornes inférieures de la fonction). Cette tâche n'est pas évidente vu que la représentation graphique d'une fonction est impossible si elle comporte plus de deux variables.

De plus, l'inexistence de méthodes analytiques (comme le calcul de dérivée) rapides pour trouver des solutions à ce genre de problèmes donne importance aux méta-heuristiques de l'optimisation continue en général.

La difficulté par rapport aux problèmes d'optimisation combinatoire, c'est qu'une variable (dimension du problème) peut prendre une infinité de valeurs possibles dans son domaine de définition, ce qui va rendre son voisinage étroit. Donc, il n'est pas pratique d'utiliser les mêmes démarches que pour un algorithme d'optimisation combinatoire pour trouver la solution souhaitée.

Pour pallier ce problème, il nous faut définir de nouvelles techniques adéquates. Notons que toutes les valeurs présentées des paramètres de l'algorithme ont été obtenues après plusieurs expérimentations.

III.2.1 Voisinage d'une solution

Soit $[a, b]^n$ le domaine de définition d'une fonction d'optimisation continue à n variables. Pour définir le voisinage d'une solution s, nous nous sommes inspirés de la méthode de N. Hu [25] qui se base sur le principe de disque.

Soit $B(s, r)$ le disque centré par s avec un rayon r, il contient l'ensemble des points s' tels que $\|s' - s\| \leq r$. Le symbole $\| \cdots \|$ est utilisé pour dénoter la norme euclidienne.

Pour une exploration homogène de l'espace de recherche, N. Hu considère un ensemble de disques centrés par la solution s, avec des rayons respectifs $h_0, h_1, ..., h_k$. Par conséquent, l'espace est partitionné en k couronnes concentriques :

$$C_j(s, h_{j-1}, h_j) = \{s' | h_{j-1} \leq \|s' - s\| \leq h_j\} \quad , \quad 1 \leq j \leq k$$

.

A partir de ces couronnes, k voisins de s sont extraits, chacun à partir d'une couronne C_j, et cela en faisant une sélection aléatoire d'un point à l'intérieur de chaque couronne C_j, pour j variant de 1 à k.

Le rayon h_k du plus grand disque est un paramètre de l'algorithme. Les autres rayons h_j $(1 \leq j \leq k-1)$ sont calculés à partir de h_k, selon la méthode de partitionnement géométrique suivante :

$$h_{k-j+1} = \frac{h_k}{2^{j-1}}, j = 2, ..., k.$$

Le rayon h_0 est un autre paramètre de l'algorithme qui satisfait la condition :

$$h_0 < \frac{h_k}{2^{k-1}}.$$

Le partitionnement de l'espace autour de s dépend donc du nombre k de voisins, du rayon h_k du disque externe et du rayon h_0 du disque interne.

Dans CBSO, nous avons adopté le principe de calcul des rayons $h_0, h_1, ..., h_k$, mais nous traitons chaque variable de la solution s indépendamment des autres. C'est pourquoi ques nous devons d'abord introduire le voisinage d'une variable.

- **Voisinage d'une variable**

Soient X_i la $i^{ème}$ variable de la solution s et x_i la valeur de cette variable. Notons cette variable (X_i, x_i). Le partitionnement du voisinage de s pour la variable X_i se fait comme suit :

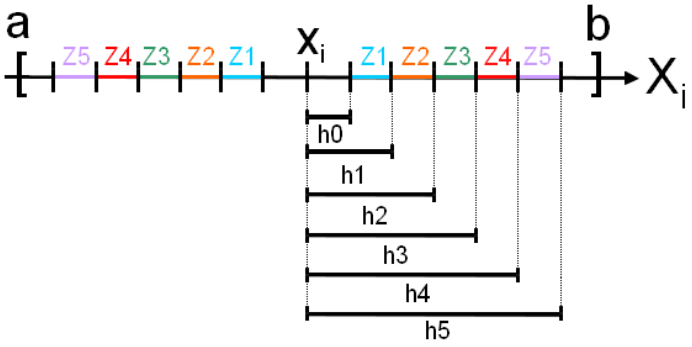


Figure III.1. Partitionnement du voisinage dans CBSO

Une zone uni-dimensionnelle Z_j définie par les rayons h_{j-1} et h_j est composée de deux parties égales, une à gauche de x_i et une à droite de x_i.

Une variable (X_j, x_j) est voisine de la variable (X_i, x_i) si et seulement si elle appartient à une des k zones uni-dimensionnelles. Autrement dit :

$$|x_i - x_j| \leq h_k$$

Une variable (X_j, x_j) n'est pas voisine de la variable (X_i, x_i) si et seulement si elle n'appartient à aucune des k zones uni-dimensionnelles. Autrement dit :

$$|x_i - x_j| > h_k$$

- **Voisinage d'une solution**

Soit V le symbole dénotant la notion de voisinage. Une solution s' est voisine de la solution s si et seulement si toutes les variables de s' sont voisines de celles de s :

$$s' \in V(s) \Leftrightarrow x'_i \in V(x_i), \forall i = 1, ..., n$$

Une solution s' n'est pas voisine de la solution s si et seulement si toutes les variables de s' ne sont pas voisines de celles de s :

$$s' \notin V(s) \Leftrightarrow x'_i \notin V(x_i), \forall i = 1, ..., n$$

Il est important de s'assurer que les coordonnées de la solution voisine ou non-voisine de s appartiennent au domaine de définition [a,b]. Pour cela, il faut s'assurer que la plus grande zone ne dépasse pas le domaine de définition de la fonction. Autrement dit, le paramètre h_k doit vérifier la contrainte suivante :

$$h_k < \frac{b-a}{2}.$$

- **Détermination d'une solution voisine**

Cela consiste à déterminer les n coordonnées d'une solution s' (une par une) dans la zone définie par les rayons h_{j-1} et h_j.

Dans le cas où la valeur x_i de la variable X_i se trouve au milieu du domaine $[a, b]$, il suffit que le générateur de nombres aléatoires choisisse une valeur x'_i, telle que :

$$x'_i \in [x_i - h_{j+1}, x_i - h_j] \cup [x_i + h_j, x_i + h_{j+1}]$$

où :

— x_i est la $i^{ème}$ coordonnée de la solution s,

— x'_i est la $i^{ème}$ coordonnée de la solution voisine.

Il se peut que la valeur x_i de la variable X_i soit proche de l'une des deux frontières du domaine de définition de la fonction (Figure III.2). Dans ce cas, une partie (hachurée) des k zones débordera du domaine [a,b], et seulement l'autre partie (non-hachurée) sera valide et pourra être utilisée pour générer la coordonnée x'_i de la solution non voisine.

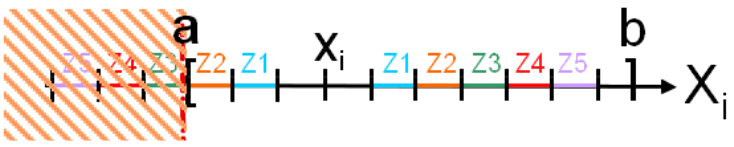


Figure III.2. Débordement du voisinage d'une variable

Si les zones débordent de la borne inférieure, le générateur de nombres aléatoires choisit la valeur x'_i telle que :

$$x'_i \in [x_i + h_j, x_i + h_{j+1}]$$

Si les zones débordent de la borne supérieure, le générateur de nombres aléatoires choisit la valeur x'_i telle que :

$$x'_i \in [x_i - h_{j+1}, x_i - h_j]$$

- **Détermination d'une solution non voisine**

Cela consiste à déterminer les coordonnées d'une solution en dehors des k zones.

Dans le cas où la valeur x_i de la variable X_i se trouve au milieu du domaine $[a, b]^2$, il suffit que le générateur de nombres aléatoires choisisse une valeur x'_i, telle que :

$$x'_i \in [a, x_i - h_k] \cup [x_i + h_k, b], \tag{III.1}$$

Si les zones débordent de la borne inférieure, alors le générateur de nombres aléatoires choisit la valeur x'_i telle que :

$$x'_i \in [x_i + h_k, b], \tag{III.2}$$

Si les zones débordent de la borne supérieure, le générateur de nombres aléatoires choisit la valeur x'_i telle que :

$$x'_i \in [a, x_i - h_k], \tag{III.3}$$

III.2.2 Distance entre deux solutions

Nous avons besoin de cette notion pour le calcul de diversité lors de la phase de diversification. Pour répondre à notre besoin, nous avons choisi la norme $infini$ qui considère le maximum des différences entre les coordonnées des deux solutions. Elle est donnée par la formule :

$$Distance(s, s') = \|s - s'\|_\infty = Max\{|x_{1i} - x_{2i}|\}, \quad 1 \leq i \leq n$$

où :

— $\|\cdots\|_\infty$ est la norme "$infini$",

— s et s' sont deux solutions,

— n est la dimension de la fonction problème (le nombre de variables),

— x_{1i} est la $i^{ème}$ variable de la première solution,

— x_{2i} est la $i^{ème}$ variable de la deuxième solution.

Par cela, nous définissons notre propre manière de voir la distance entre deux solutions.

III.2.3 Diversité d'une solution

C'est une valeur réelle que nous calculons pour chaque solution de la table *Danse*. Elle est donnée par la formule suivante :

$$Diversité(s) = Min\{\|s - s'\|_\infty, \forall s' \in Tabou\}.$$

Cette mesure permet d'indiquer à quel point la solution s est éloignée des sommets de référence des itérations précédentes. Rappelons que la liste $Tabou$ contient les anciens sommets de référence.

III.2.4 Algorithme

Algorithme III.2 : CBSO

Entrées : Les paramètres de l'algorithme et la condition d'arrêt
Sorties : La meilleure solution trouvée

```
1  début
2      Soit Sref la solution trouvée par l'abeille éclaireuse;
3      tant que la condition d'arrêt est non vérifiée faire
4          Insérer Sref dans la liste Tabou;
5          Déterminer la zone de recherche à partir de Sref;
6          Attribuer une solution s de la zone de recherche à chaque abeille;
7          pour chaque abeille faire
8              Recherche locale avec la solution s;
9              Mettre l'optimum local dans la table Danse;
10         fin pour
11         Construire la solution noyau et l'insérer dans la table Danse;
12         Choisir le nouveau sommet de référence Sref;
13     fin tq
14 fin
```

Nous pouvons constater que notre algorithme ne change pas la structure générale de l'algorithme BSO sauf que chaque étape est définie de manière à être mieux adaptée aux problèmes continus.

- **Génération du sommet de référence initial**

Le sommet de référence initial est construit d'une manière incrémentale (variable par variable). A chaque variable de $Sref$, le générateur de nombres aléatoires attribue une valeur réelle aléatoire qui obéit à la loi uniforme donnée par la formule suivante :

$$u(x) = \begin{cases} \frac{1}{b-a} & \text{si} \quad x \in [a,b] \\ 0 & \text{sinon} \end{cases} \tag{III.4}$$

Toutes les valeurs réelles appartenant à l'intervalle $[a, b]$ ont la même probabilité ($p = \frac{1}{b-a}$) d'être choisies et aucune autre valeur ne sera choisie ($p = 0$).

- **Détermination de la zone de recherche**

Elle consiste à générer m solutions à partir du sommet de référence (m étant le nombre d'abeilles) et à attribuer une solution à chaque abeille pour démarrer la recherche locale.

Nous avons considéré quatre abeilles dont les solutions sont distribuées comme suit,

— la première abeille travaille sur le sommet de référence. Cela signifie qu'à partir de la deuxième itération, elle va continuer le travail des abeilles de l'itération précédente (*intensifier* encore plus la recherche),

— la deuxième abeille travaille sur une solution non voisine du sommet de référence, pour *diversifier* la zone de recherche. Cette solution est déterminée en utilisant une des formules III.1, III.2 ou III.3,

— En utilisant le paramètre $Flip$, la troisième abeille combine entre le sommet de référence et une solution non voisine de telle manière à obtenir une *solution hybride*. Nous avons choisi $Flip = 2$ ce qui signifie que cette abeille prend les cases impaires du sommet de référence et les cases paires de la solution non voisine.

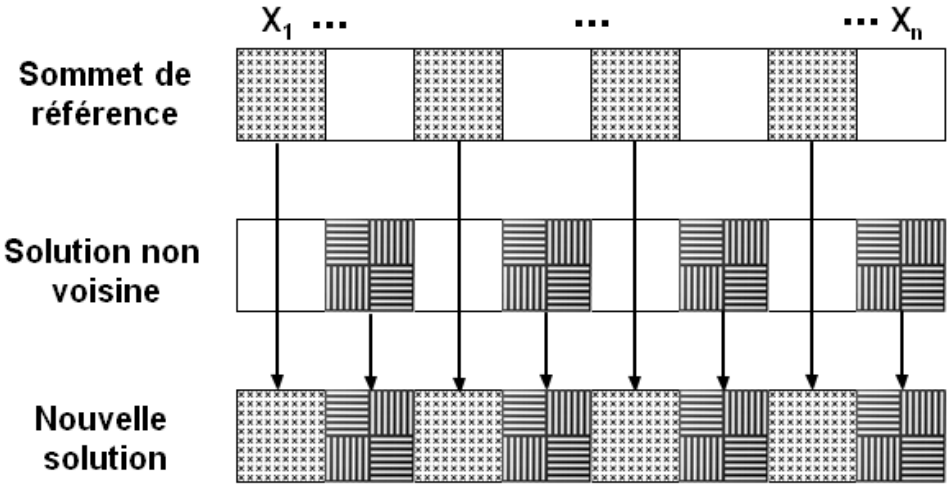


Figure III.3. Construction d'une solution à partir de Sref et d'une solution non voisine

L'algorithme de construction de la nouvelle solution hybride en utilisant le paramètre *Flip* est donné comme suit :

Algorithme III.3 : Construction de la nouvelle solution

Entrées : $Sref$, solution non voisine et $Flip$

Sorties : La nouvelle solution hybride

```
1 début
2     solutionHybride← Sref;
3     i ← Flip;
4     tant que i ≤ n faire
5         solutionHybride[i]← solutionNonVoisine[i];
6         i ← i + Flip;
7     fin tq
8 fin
```

— la quatrième abeille fait de même que pour la troisième abeille, sauf que la solution obtenue prend les cases paires du sommet de référence et les cases impaires de la solution non voisine.

- **Recherche locale**

Après avoir attribué les solutions de la zone de recherche aux abeilles, chacune d'elles démarre une recherche locale avec sa solution en explorant son voisinage pour éventuellement choisir le voisin qui minimise le plus la fonction objectif.

Après plusieurs expérimentations, nous avons choisi d'opter pour le voisinage en étoile basé sur les deux étapes suivantes :

— Générer 5 voisins ($k = 5$) tels que les coordonnées de chaque voisin sont prises d'une zone différente ($Z_1, Z_2, ..., Z_5$) pour une exploration judicieuse de l'espace de recherche.

— Comparer les k voisins générés deux à deux, puis prendre le meilleur et le comparer avec la solution s. S'il est meilleur, il sera considéré comme l'optimum local pour cette abeille à cette itération, sinon, la solution s le sera.

Le processus de recherche locale est répété un nombre de fois défini par le paramètre *SearchIter*.

Algorithme III.4 : Recherche locale

Entrées : Solution s, *SearchIter* et k
Sorties : Optimum local

```
1  début
2      pour i allant de 1 à SearchIter faire
3          pour j allant de 1 à k faire
4              Générer un voisin v_i dans la j^ième zone;
5              si fitness(v_i)← optimum alors
6                  Retourner v_i;
7              sinon si v_i est le meilleur voisin alors
8                  meilleur_voisin ← v_i;
9              fin si
10         fin pour
11         si fitness(meilleur_voisin) ≤ fitness(s) alors
12             Retourner meilleur_voisin;
13         sinon
14             Retourner s;
15         fin si
16     fin pour
17 fin
```

A la fin de la recherche locale, chaque abeille communique son optimum local à travers la table *Danse*. Celle-ci va donc contenir à la fin de chaque itération, les sommets de référence potentiels de la prochaine itération.

- **Construction de la solution noyau**

En plus des m optimums locaux existant dans la table *Danse*, nous construisons, à partir de la même table, une autre solution que nous appelons *solution noyau*. Nous nous sommes inspirés de $ACO_{\mathbb{R}}$ [22] pour construire cette solution en faisant un choix probabiliste sur ses n composants.

Pour cela, nous utilisons la fonction gaussienne noyau (Gaussian kernel) qui est une distribution de probabilités continue construite à partir de n fonctions gaussiennes individuelles associées aux solutions de la table *Danse*.

Pour chaque dimension i du problème, il existe une fonction gaussienne noyau G^i différente dont la formule est la suivante :

$$G^i(x) = \sum_{j=1}^{m} \omega_j \frac{1}{\sigma_j^i \sqrt{2\pi}} \mathrm{e}^{-\frac{(x-\mu_j^i)^2}{2\sigma_j^{i2}}},$$

où :

— m est le nombre de solutions dans la table *Danse*,

— ω_j est le poids associé à la $j^{ème}$ solution donné par son rang (la solution qui a la meilleure évaluation aura le plus grand poids et celle qui a la pire évaluation aura le plus petit poids),

— μ^i est le vecteur des moyennes,

— σ^i est le vecteur des écart-types.

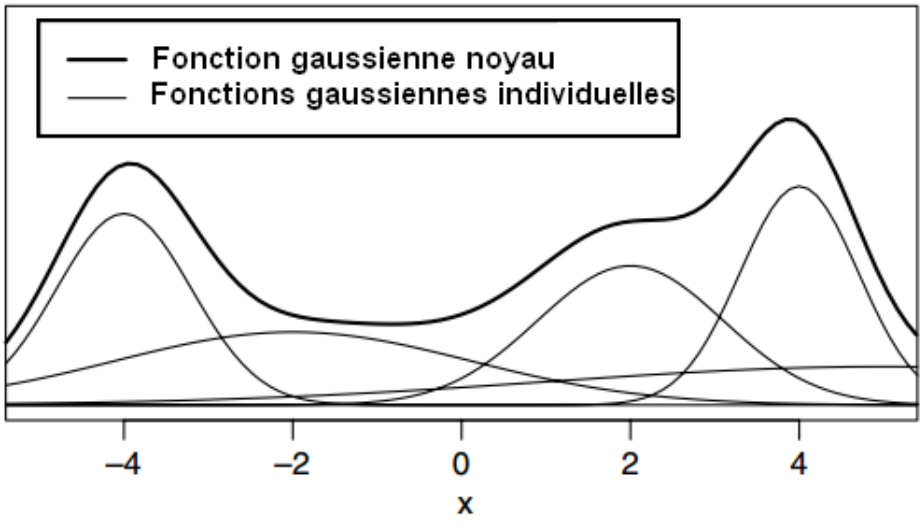


Figure III.4. Représentation graphique de la fonction gaussienne noyau [22]

La fonction gaussienne noyau nous permet de détecter les régions prometteuses à partir des optimums locaux trouvés par les abeilles. Cela nous aide à nous rapprocher encore de l'optimum global.

Le processus de construction de la nouvelle solution est appelé échantillonnage de la fonction gaussienne noyau et est défini pratiquement, pour chaque dimension i du problème, par :

— le choix de la meilleure solution s_l dans la table *Danse*,

— le calcul des valeurs μ_l^i et σ_l^i associées à s_l,

— l'appel au générateur de nombre aléatoires gaussien qui prend en entrée μ_l^i et σ_l^i et retourne une valeur réelle aléatoire x qui obéit à la loi gaussienne $N(\mu_l^i, \sigma_l^i)$ donnée par la formule :

$$g(x) = \frac{1}{\sigma_l^i\sqrt{2\pi}} \mathrm{e}^{-\frac{(x-\mu_l^i)^2}{2{\sigma_l^i}^2}} \quad , \quad x \in [a, b].$$

.

μ_l^i est tout simplement la $i^{ème}$ valeur de la meilleure solution ($\mu_l^i = s_l^i$).

L'écart-type σ_l^i représente une erreur moyenne entre la $i^{ème}$ valeur de s_l et celles des autres solutions et se calcule par la formule suivante :

$$\sigma_l^i = \xi \sum_{j=1}^{m} \frac{|S_j^i - S_l^i|}{m-1}.$$

où ξ est le paramètre qui contrôle la vitesse de convergence de la solution noyau.

Après la construction de la solution noyau, nous l'évaluons et l'insérons dans la table *Danse*, puis nous choisissons le prochain sommet de référence.

- **Choix du nouveau sommet de référence**

Lorsqu'il s'agit des fonctions multi-modales (à plusieurs optimums locaux), il est très important que l'algorithme évite d'être stagné dans un optimum local. Pour cela, il faut utiliser une stratégie de diversification de recherche afin de pouvoir converger vers l'optimum global.

En réalité, ce sont deux buts contradictoires. D'une part, un algorithme est supposé converger le plus vite possible et d'autre part, il est supposé ne pas converger entièrement vers un optimum local.

Cette contradiction est due au fait q'un algorithme ne peut pas savoir si une région prometteuse mène à un optimum local ou à un optimum global. Le défi est d'établir une

balance entre l'efficacité (diversification) et la rapidité (intensification). Cela se traduit par la capacité de choisir le bon prochain sommet de référence à partir de la table *Danse* qui contient $m + 1$ solutions.

Soient :

— $Sref_t$ le sommet de référence à l'instant t,

— $Sref_{t+1}$ le sommet de référence à l'instant $t + 1$,

— $MSolutionQualité$ la meilleure solution en qualité à l'instant t,

— $MSolutionDiversité$ la meilleure solution en diversité à l'instant t.

Algorithme III.5 : Choix du nouveau sommet de référence

Entrées : La table *Danse*

Sorties : Le nouveau sommet de référence $Sref_{t+1}$

```
1  début
2      si fitness(MSolutionQualité) ≤ fitness(Sref_t)/* La meilleure solution à
           cette itération est meilleure que Sref_t */
3       alors
4           Sref_{t+1} ← MSolutionQualité;
5           NbChances ← MaxChances;
6      sinon
7           /* La meilleure solution à cette itération est pire que
              Sref_t */
8           NbChances ← NbChances − 1;
9          si NbChances > 0 alors
10             Sref_{t+1} ← MSolutionQualité;
11         sinon
12             Sref_{t+1} ← MSolutionDiversité;
13             NbChances ← MaxChances;
14         fin si
15     fin si
16 fin
```

Si la meilleure solution en qualité (celle qui a la plus petite évaluation) est meilleure que le sommet de référence, elle sera considérée comme le prochain sommet de référence. Sinon, elle le sera quand même sauf que nous décrémentons le nombre de chances. Si le nombre de chances devient nul, la meilleure solution en diversité (celle qui a la plus grande diversité) sera considérée comme le prochain sommet de référence et le nombre de chances sera réinitialisé à son maximum.

— Lors du choix de la meilleure solution en qualité (instructions 4 et 10), si deux solutions ont la même qualité, celle qui a la plus grande diversité sera considérée.

— Lors du choix de la meilleure solution en diversité (instruction 12), si deux solutions ont la même diversité, celle qui a la plus petite évaluation (meilleure qualité) sera considérée.

Il est important que la meilleure solution (en qualité ou en diversité) choisie ne soit pas taboue. Si elle l'est, nous choisissons la prochaine meilleure solution et ainsi de suite. Il peut arriver, même rarement, que toutes les solutions de la table *Danse* soient taboues. Dans ce cas, le prochain sommet de référence sera généré aléatoirement selon la loi uniforme (équation III.4).

- **Condition d'arrêt**

Les instructions 5 à 12 de l'algorithme CBSO sont répétées jusqu'à ce qu'une de ces trois conditions soient vérifiées :

— l'algorithme converge vers la solution souhaitée ($|f - f^*| < \epsilon$ où ϵ est un paramètre de l'algorithme),

— le nombre maximum des itérations est atteint,

— une stagnation est détectée (on n'aperçoit pas de changement dans le sommet de référence au bout d'un nombre d'itérations défini par le paramètre *Stag*).

Conclusion

Nous avons présenté dans ce chapitre l'adaptation de la méta-heuristique BSO aux problèmes d'optimisation continue. Nous passons à présent à l'implémentation de notre algorithme.

Chapitre IV

Implémentation de l'algorithme

Introduction

Dans ce chapitre, nous expliquons l'environnement de travail que nous avons utilisé afin d'implémenter CBSO, puis nous présentons l'interface graphique de notre application.

IV.1 Outils de travail

Nous avons eu besoin des outils suivants :

— langage de programmation Java,

— environnement de développement Netbeans,

— machine dotée d'un processeur Intel core i3 et d'une RAM de 4GO.

IV.2 Application

L'interface graphique de notre application aide l'utilisateur à paramétrer l'algorithme et à l'exécuter sur les différentes fonctions de test.

Les paramètres de l'algorithme sont chargés automatiquement à l'apparition de l'interface graphique, ils peuvent être changés par l'utilisateur.

Les figures suivantes montrent l'interface graphique avec les différentes étapes d'utilisation.

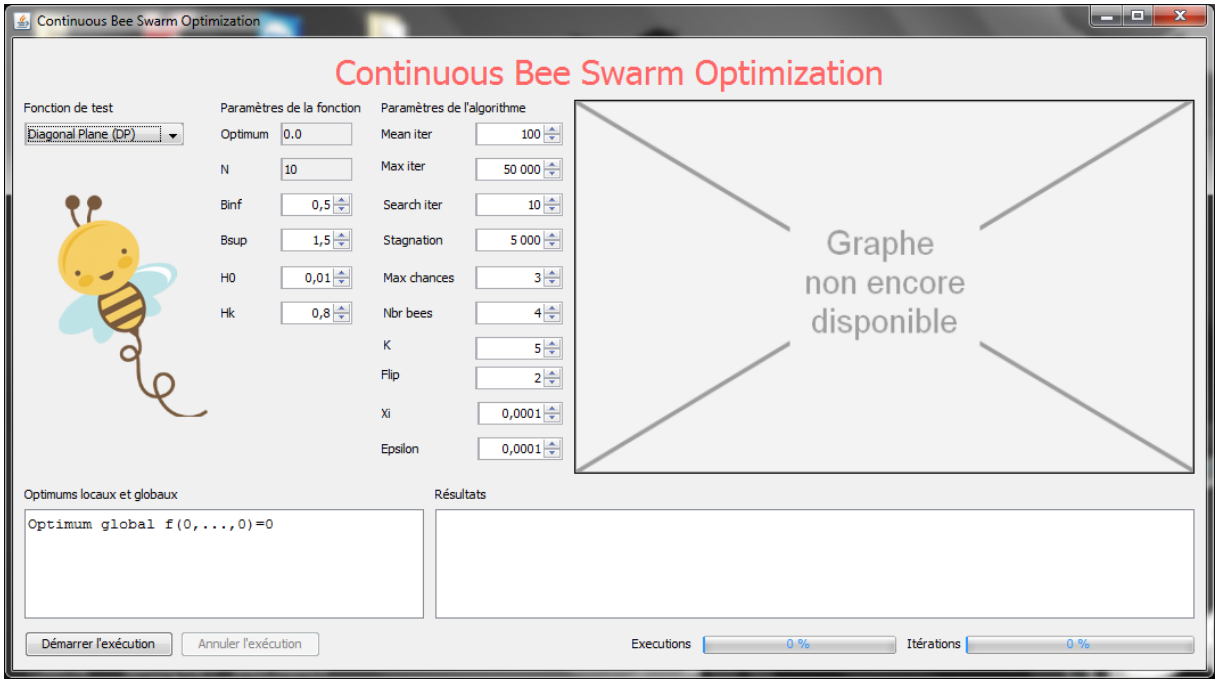


Figure IV.1. Interface graphique de l'application

Il suffit que l'utilisateur passe le curseur sur le nom d'un paramètre pour qu'une information explicative de ce dernier apparaisse.

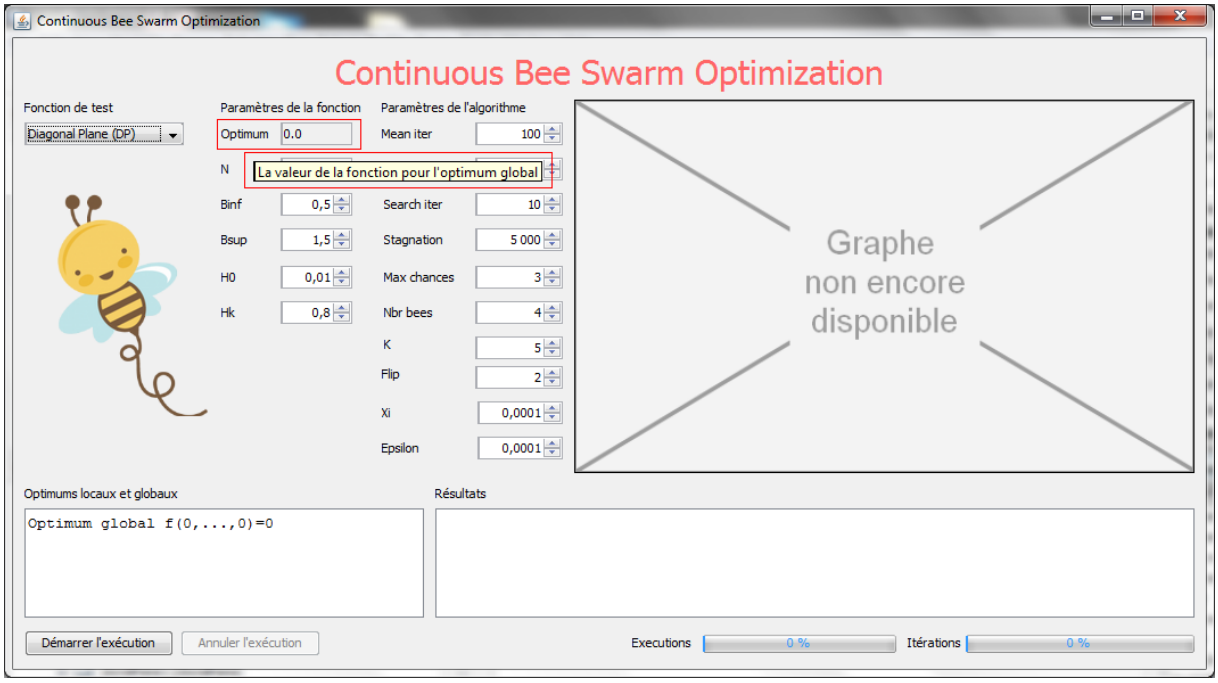


Figure IV.2. Explication des paramètres de l'algorithme

L'utilisateur commence par choisir la fonction de test à partir de la liste déroulante qui se trouve en haut à gauche de l'interface.

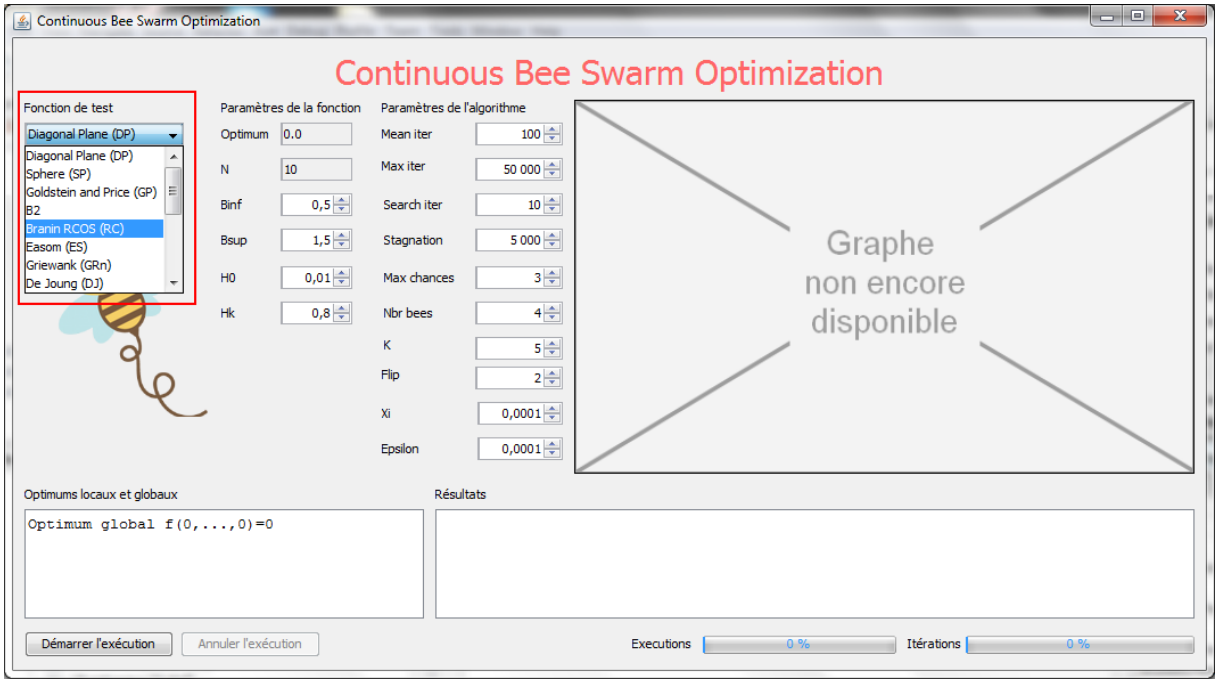


Figure IV.3. Choix de la fonction de test

Après avoir choisi la fonction, ses paramètres sont chargés automatiquement. L'utilisateur peut choisir de travailler sur un sous-domaine du domaine de définition de la fonction comme il peut changer les paramètres h_0 et h_k.

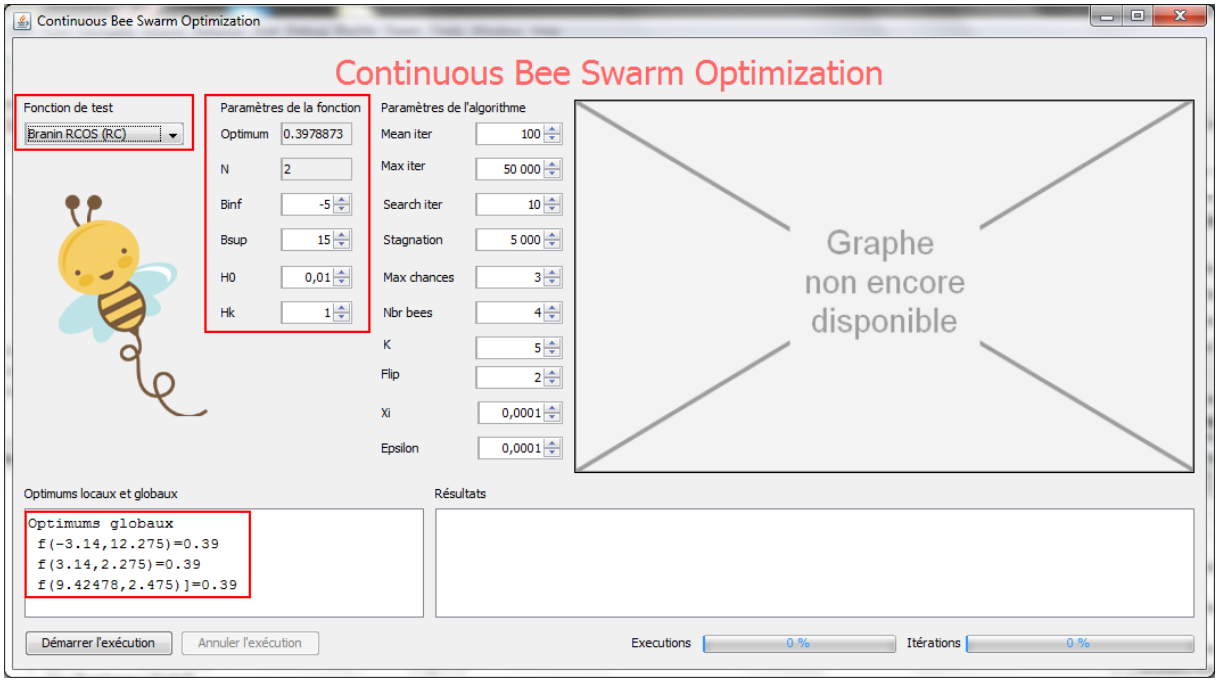


Figure IV.4. Chargement des paramètres de la fonction de test

Avant de lancer CBSO, l'utilisateur peut fixer le nombre d'exécutions indépendantes de l'algorithme à travers le paramètre *MeanIter*. Nous avons fixé par défaut ce nombre à 100 itérations indépendantes et cela pour avoir des résultats plus fiables.

L'utilisateur peut alors lancer CBSO en appuyant sur le bouton *Démarrer l'exécution*. La progression de l'algorithme est observée à travers deux barres de progression, la première indiquant le numéro de l'exécution courante et la deuxième le numéro d'itération courante dans l'exécution.

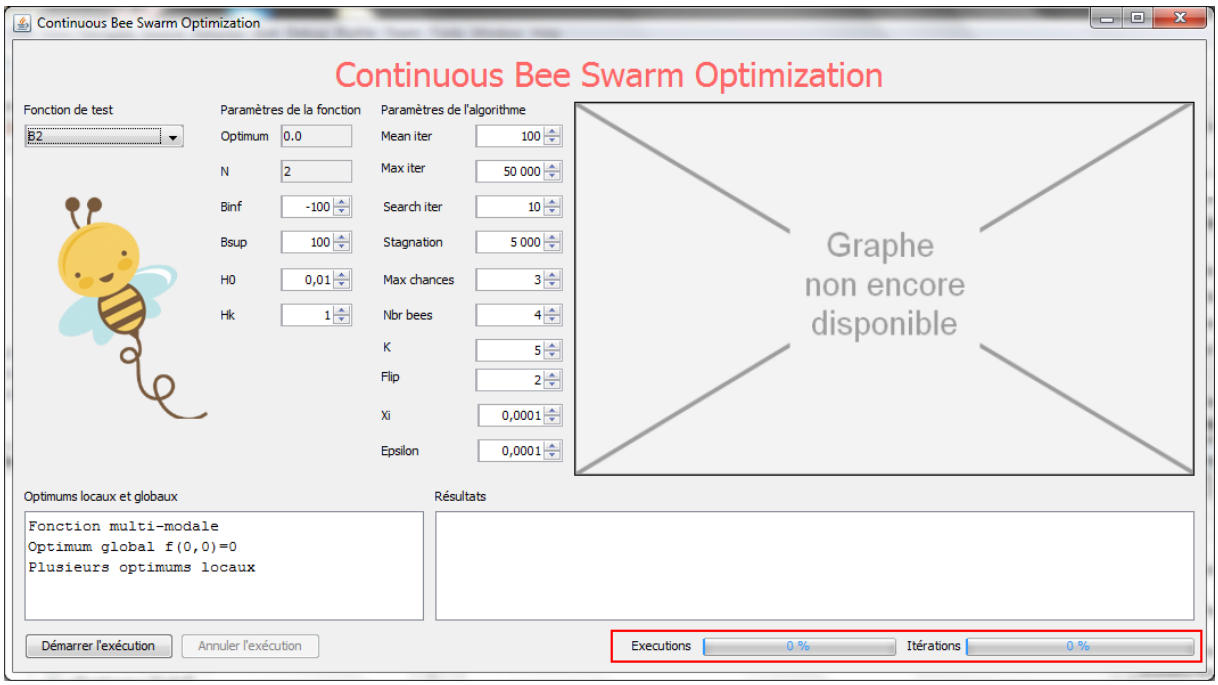


Figure IV.5. Barres de progression

Dans le cas où l'utilisateur veut interrompre l'exécution, il appuie sur le bouton *Annuler l'exécution*.

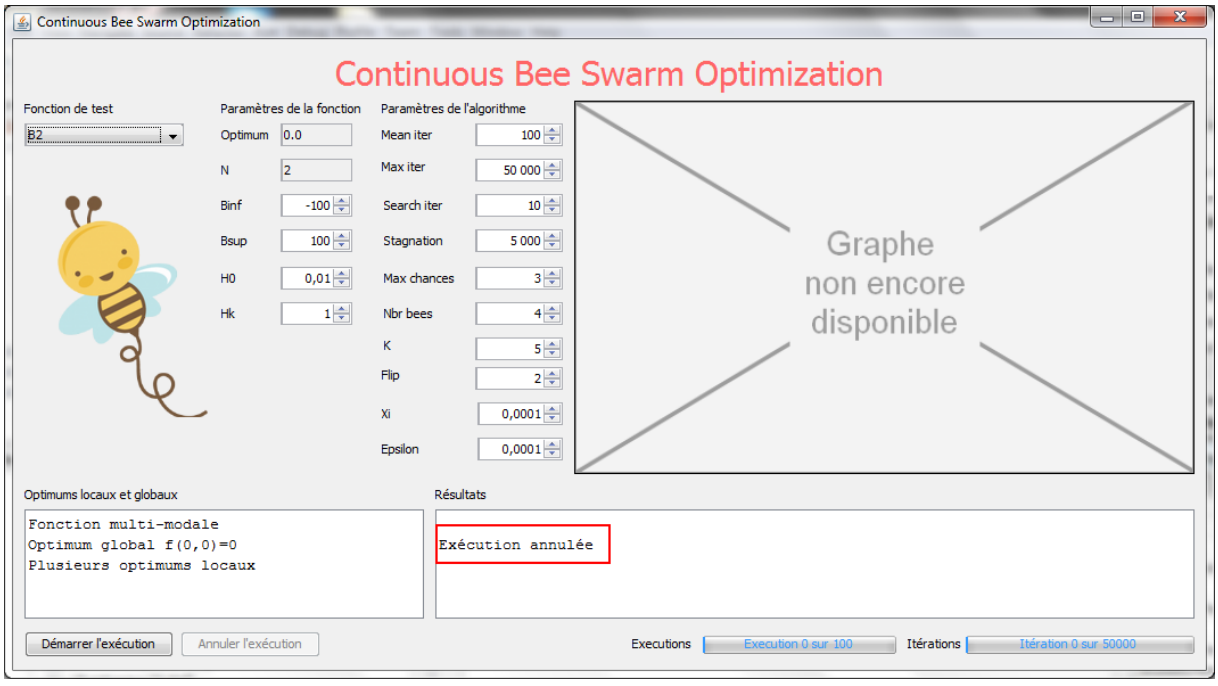


Figure IV.6. Annulation de l'exécution

A la fin de l'algorithme, les informations concernant l'exécution sont affichées dans la zone de texte en bas à droite. Ces informations comportent la meilleure solution trouvée, le nombre des itérations, le nombre des évaluations de la fonction, le taux de réussite, de stagnation et d'atteinte de $MaxIter$ ainsi que le temps d'exécution.

De plus, un graphe est affiché représentant l'évolution de l'évaluation du sommet de référence à travers les 3000 premières itérations au maximum de la dernière exécution.

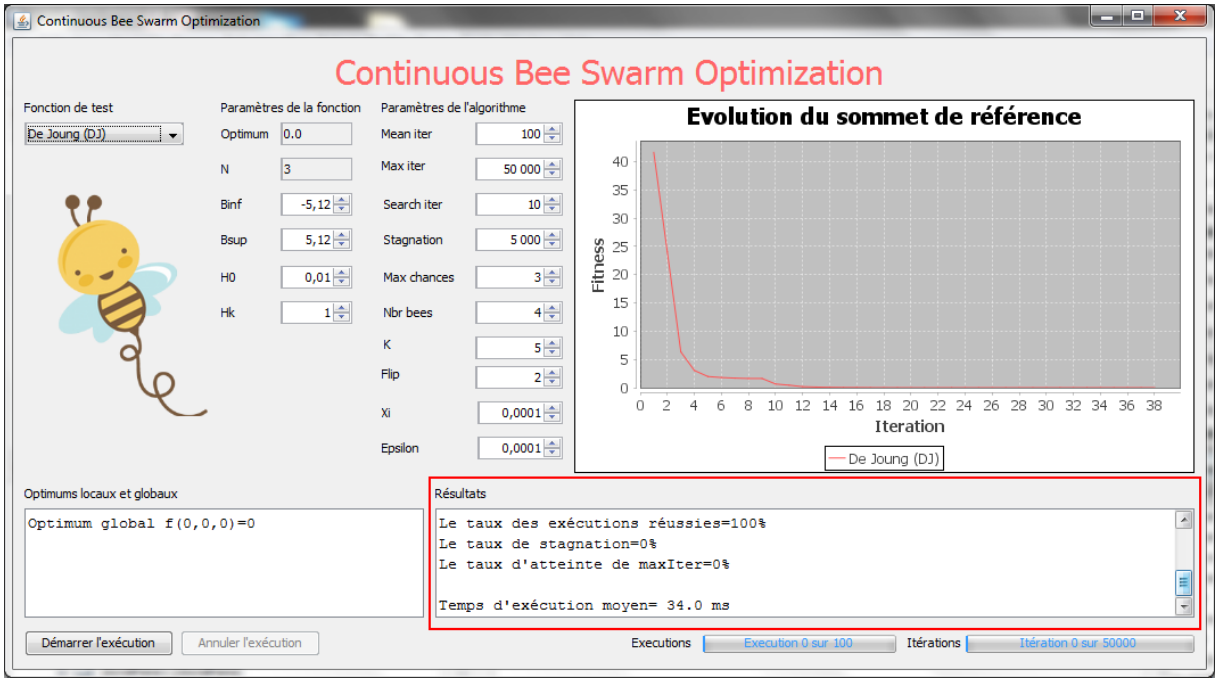


Figure IV.7. Affichage des résultats

Conclusion

Après avoir présenté l'interface graphique de l'application, l'étape suivante est celle du test et de l'évaluation de CBSO.

Chapitre V

Expérimentations et résultats

Introduction

En général, il y a deux catégories de problèmes d'optimisation, les fonctions de test et les problèmes du monde réel. Les fonctions de test sont des problèmes artificiels qui peuvent être utilisés pour l'évaluation d'un algorithme dans différentes situations avec plusieurs niveaux de difficulté.

Les problèmes du monde réel touchent à différents domaines tels la physique, la chimie, l'ingénierie et les mathématiques. Ces problèmes sont plus difficiles à manipuler que les fonctions de test artificielles, mais ils se ramènent toujours à une ou à plusieurs d'entre elles.

Après l'implémentation de CBSO, nous pouvons donner les fonctions de test utilisées ainsi que les paramètres de l'algorithme. Ensuite, nous présentons les résultats de nos expérimentations qui sont une phase très importante pour l'évaluation de notre travail. Enfin, nous faisons une étude comparative entre CBSO et d'autres algorithmes d'optimisation continue en vue d'examiner les performances de CBSO et sa capacité de converger vers l'optimum global face aux fonctions multimodales.

V.1 Fonctions de test

Nous avons mené nos expérimentations sur des benchmarks publics utilisés par des méthodes avec lesquelles nous envisageons une comparaison. Ces benchmarks contiennent des fonctions de test de plusieurs types dont les optimums globaux sont connus.

Les fonctions multi-modales avec plusieurs optimums locaux sont parmi les classes de problèmes les plus difficiles, tandis que les fonctions à surfaces plates posent une difficulté à l'algorithme parce qu'elles ne donnent pas d'information pour le diriger vers l'optimum.

La liste des fonctions de test que nous avons utilisées est la suivante[24].

— Diagonal Plane (DP)

$$DP(\overrightarrow{x}) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- n =10,
- $\overrightarrow{x} \in [0.5, 1.5]^n$,
- $h_0 = 10^{-22}$,
- $h_k = 7 * 10^{-21}$,
- optimum global DP(0.5,...,0.5)=0.5,
- continue, non-séparable et uni-modale.

— Sphere (SP)

$$SP(\overrightarrow{x}) = \sum_{i=1}^{n} x_i^2$$

- n =10,
- $\overrightarrow{x} \in [-3, 7]^n$,
- $h_0 = 10^{-5}$,
- $h_k = 10^{-1}$,
- optimum global SP(0,...,0)=0,
- continue, séparable et multi-modale.

— B_2

$$B_2(\overrightarrow{x}) = (x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1) - 0.4\cos(4\pi x_2) + 0.7)$$

- n =2,
- $\overrightarrow{x} \in [-100, 100]^n$,
- $h_0 = 3 * 10^{-3}$,
- $h_k = 2$,
- optimum global B_2(0,0)=0,
- continue, non-séparable et multi-modale.

— Branin RCOS (RC)

$$RC(\overrightarrow{x}) = \left(x_2 - \left(\frac{5}{(4\pi^2)}\right)x_1^2 + \left(\frac{5}{\pi}\right)x_1 - 6\right)^2 + 10\left(1 - \left(\frac{1}{8\pi}\right)\right)\cos(x_1) + 10$$

- n =2,
- $\overrightarrow{x} \in [-5, 15]^n$,
- $h_0 = 6 * 10^{-4}$,
- $h_k = 8 * 10^{-1}$,
- optimums globaux RC(-π,12.275)=0.39, RC(π,2.275)=0.39, RC(9.42478,2.475)=0.39,
- continue, non-séparable et multi-modale.

— Easom (ES)

$$ES(\overrightarrow{x}) = -\cos(x_1)\cos(x_2)\exp(-((x_1 - \pi)^2 + (x_2 - \pi)^2))$$

- n =2,
- $\overrightarrow{x} \in [-100, 100]^n$,
- $h_0 = 8 * 10^{-4}$,
- $h_k = 2$,
- optimum global ES(π,π)=-1,
- continue, séparable et multi-modale.

— De Joung (DJ)

$$DJ(\overrightarrow{x}) = x_1^2 + x_2^2 + x_3^2$$

- n =3,
- $\overrightarrow{x} \in [-5.12, 5.12]^n$,
- $h_0 = 3 * 10^{-4}$,
- $h_k = 4 * 10^{-1}$,
- optimum global DJ(0,0,0)=0,
- continue, séparable et multi-modale.

— Rosenbrock2 (R_2)

$$R_n(\overrightarrow{x}) = \sum_{i=1}^{n-1}[100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2]$$

- n =2,
- $\overrightarrow{x} \in [-5, 10]^n$,
- $h_0 = 10^{-2}$,
- $h_k = 1$,
- optimum global R_n(1,...,1)=0,
- continue, non-séparable et uni-modale.

— Zakharov2 (Z_2)

$$Z_n(\overrightarrow{x}) = \left(\sum_{i=1}^{n} x_i^2\right) + \left(\sum_{i=1}^{n} 0.5ix_i\right)^2 + \left(\sum_{i=1}^{n} 0.5ix_i\right)^4$$

- n =2,
- $\overrightarrow{x} \in [-5, 10]^n$,
- $h_0 = 10^{-2}$,
- $h_k = 2 * 10^{-1}$,
- optimum global Z_n(0,...,0)=0,
- continue, non-séparable et multi-modale.

— Zakharov5 (Z_5)

$$Z_n(\overrightarrow{x}) = \left(\sum_{i=1}^{n} x_i^2\right) + \left(\sum_{i=1}^{n} 0.5ix_i\right)^2 + \left(\sum_{i=1}^{n} 0.5ix_i\right)^4$$

- n =5,
- $\overrightarrow{x} \in [-5, 10]^n$,
- $h_0 = 2 * 10^{-3}$,
- $h_k = 2 * 10^{-1}$,
- optimum global Z_n(0,...,0)=0,
- continue, non-séparable et multi-modale.

— Hartmann$_3$ ($H_{3,4}$)

$$H_{3,4}(\overrightarrow{x}) = -\sum_{i=1}^{4} c_i \exp[-\sum_{j=1}^{3} a_{ij}(x_j - p_{ij})^2]$$

- n =3,
- $\overrightarrow{x} \in [0,1]^n$,
- $h_0 = 6 * 10^{-5}$,
- $h_k = 8 * 10^{-2}$,
- optimum global $H_{3,4}$(0.1140,0.556,0.852)=-3.86,
- continue, non-séparable et multi-modale (4 optimums locaux).

a_{ij}			c_i	p_{ij}		
3.0	10.	30.	1.0	0.3689	0.1170	0.2673
0.1	10.	35.	1.2	0.4699	0.4387	0.7470
3.0	10.	30.	3.0	0.1091	0.8732	0.5547
0.1	10.	35.	3.2	0.0381	0.5743	0.8828

— Hartmann$_6$ ($H_{6,4}$)

$$H_{6,4}(\overrightarrow{x}) = -\sum_{i=1}^{4} c_i \exp[-\sum_{j=1}^{6} a_{ij}(x_j - p_{ij})^2]$$

- n =6,
- $\overrightarrow{x} \in [0,1]^n$,
- $h_0 = 10^{-3}$,
- $h_k = 2 * 10^{-2}$,
- optimum global $H_{6,4}$(0.2016,0.1500,0.4768,0.2753,0.3116,0.6573)=-3.32,
- continue, non-séparable et multi-modale (4 optimums locaux).

a_{ij}						c_i	p_{ij}					
10.	3.	17.	3.5	1.7	8.	1.0	0.1312	0.1696	0.5569	0.0124	0.8283	0.5886
0.05	10.	17.	0.1	8.0	14.0	1.2	0.2329	0.4135	0.8307	0.3736	0.1004	0.9991
3.0	3.5	1.7	10.0	17.0	8.0	3.0	0.2348	0.1451	0.3522	0.2883	0.3047	0.6650
17.0	8.0	0.05	10.0	0.1	14.0	3.2	0.4047	0.8828	0.8732	0.5743	0.1091	0.0381

— Shekel$_5$ ($S_{4,5}$)

$$S_{4,5}(\overrightarrow{x}) = -\sum_{i=1}^{5}[(\overrightarrow{x} - a_i)^T(\overrightarrow{x} - a_i) + c_i]^{-1}$$

- n =4,
- $\overrightarrow{x} \in [0, 10]^n$,
- $h_0 = 6 * 10^{-5}$,
- $h_k = 2 * 10^{-2}$,
- optimum global $S_{4,5}$(4,4,4,4)=-10.14,
- continue, non-séparable et multi-modale (5 optimums locaux).

a_{ij}				c_i
4	4	4	4	0.1
1	1	1	1	0.2
8	8	8	8	0.2
6	6	6	6	0.4
3	7	3	7	0.4

— Shekel$_7$ ($S_{4,7}$)

$$S_{4,7}(\overrightarrow{x}) = -\sum_{i=1}^{7}[(\overrightarrow{x} - a_i)^T(\overrightarrow{x} - a_i) + c_i]^{-1}$$

- n =4,
- $\overrightarrow{x} \in [0, 10]^n$,
- $h_0 = 6 * 10^{-5}$,
- $h_k = 2 * 10^{-2}$,
- optimum global $S_{4,7}$(4,4,4,4)=-10.39,
- continue, non-séparable et multi-modale (7 optimums locaux).

a_{ij}				c_i
4	4	4	4	0.1
1	1	1	1	0.2
8	8	8	8	0.2
6	6	6	6	0.4
3	7	3	7	0.4
2	9	2	9	0.6
5	5	3	3	0.3

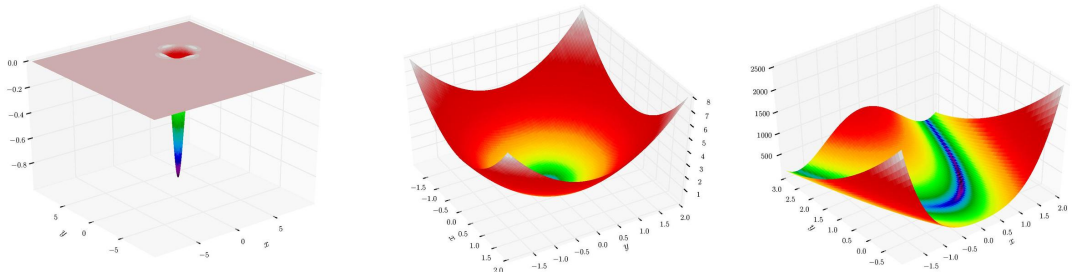
— Shekel$_{10}$ ($S_{4,10}$)

$$S_{4,10}(\overrightarrow{x}) = -\sum_{i=1}^{10}[(\overrightarrow{x} - a_i)^T(\overrightarrow{x} - a_i) + c_i]^{-1}$$

- n =4,
- $\overrightarrow{x} \in [0, 10]^n$,
- $h_0 = 6 * 10^{-5}$,
- $h_k = 2 * 10^{-2}$,
- optimum global $S_{4,10}$(4,4,4,4)=-10.53,
- continue, non-séparable et multi-modale (10 optimums locaux).

a_{ij}				c_i
4	4	4	4	0.1
1	1	1	1	0.2
8	8	8	8	0.2
6	6	6	6	0.4
3	7	3	7	0.4
2	9	2	9	0.6
5	5	3	3	0.3
8	1	8	1	0.7
6	2	6	2	0.5
7	3.6	7	3.6	0.5

Comme nous l'avons expliqué, la représentation graphique d'une fonction est impossible si elle comporte plus de deux variables. Nous donnons ici la représentation de quelques fonctions à deux variables :



(a) Easom *(b) Sphere* *(c) RosenBrock2*

Figure V.1. Représentation graphique de quelques fonctions de test à deux variables

V.2 Paramètres de CBSO

Chaque algorithme d'optimisation est caractérisé par un certain nombre de paramètres qui le définissent et le guident dans le processus de recherche de solution. La liste des paramètres que nous avons utilisés dans CBSO est la suivante.

Paramètre	Signification	Valeur
m	Nombre d'abeilles.	4
k	Nombre de voisins.	5
$SearchIter$	Nombre d'itérations d'une recherche locale.	10
ξ	Vitesse de convergence de la solution noyau.	0.0001
$Stag$	Nombre toléré d'itérations avant de détecter la stagnation.	5000
$MaxChances$	Nombre maximum de chances pour améliorer le sommet de référence.	3
$1/Flip$	Pourcentage de variables à remplacer dans le $SRef$.	1/2
$MaxIter$	Nombre maximum des itérations pour trouver la solution.	50000
ϵ	Précision voulue : différence tolérée entre l'évaluation de la solution et l'optimum global.	10^{-4}

Tableau V.1. Paramètres de CBSO

V.3 Analyse de CBSO

Comme nous l'avons précisé, l'utilisateur visualise, à partir de l'interface graphique, le graphe représentant l'évolution de l'évaluation du sommet de référence au cours des itérations. La figure suivante montre ce graphe pour la fonction *De Joung*.

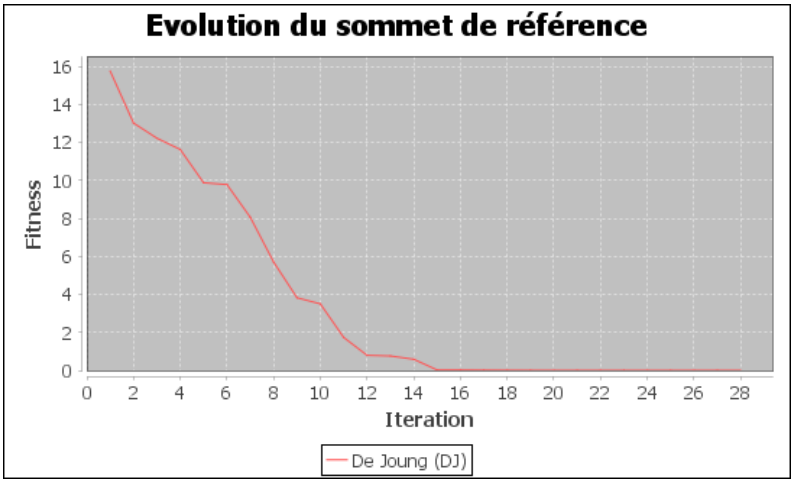


Figure V.2. Evolution de $SRef$ pour la fonction De Joung

Nous remarquons que CBSO converge bien vers l'optimum. En commençant à partir d'une solution aléatoire qui a une grande évaluation, l'algorithme se rapproche petit à petit de la solution qui a la plus petite évaluation.

Lorsqu'il s'agit des fonctions multi-modales, nous remarquons des sauts dans l'évaluation du sommet de référence dans le graphe. Ces pics indiquent les moments de diversification où l'algorithme décide à régresser pour mieux avancer. Cela se traduit par un changement de la région à explorer de l'espace de recherche.

Nous donnons comme exemple la fonction multi-modale $Shekel_{4,10}$.

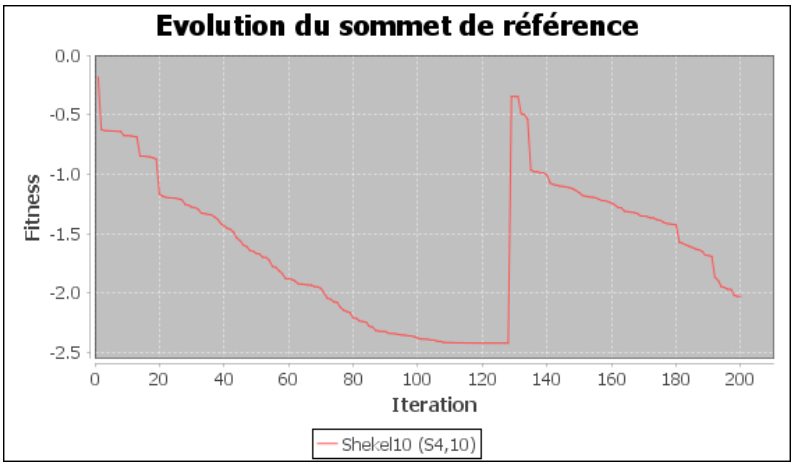


Figure V.3. Evolution de SRef pour la fonction $Shekel_{4,10}$

Pour simplifier, supposons que nous sommes face à une fonction à une seule variable. La figure suivante montre l'utilité de diversification en cas de stagnation dans un optimum local.

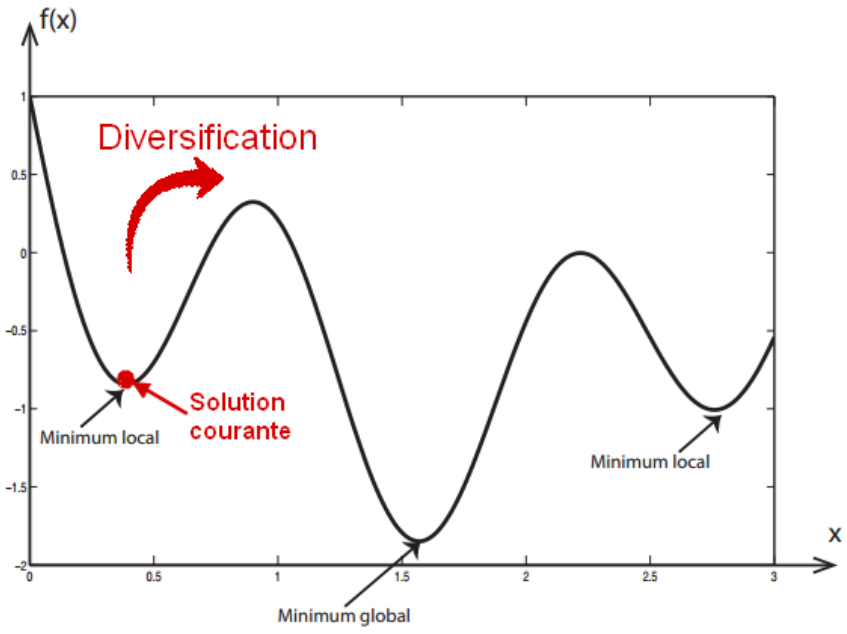


Figure V.4. Utilité de la diversification pour le changement de la zone de recherche

La diversification dans CBSO est faite lorsque le nombre de chances pour améliorer le sommet de référence devient nul, et cela par le choix de la meilleure solution en diversité à partir de la table *Danse*, c'est-à-dire la solution la plus éloignée des sommets de référence précédents. Dans le cas échéant, la diversification est faite en choisissant un sommet de référence aléatoire et cela lorsque toutes les solutions de la table *Danse* sont taboues.

Parmi les tâches les plus difficiles d'un algorithme d'optimisation, il y a celle de pouvoir diversifier au bon moment et vers la bonne région de l'espace de recherche, car une diversification aléatoire peut nous éloigner de l'optimum global et nous mener vers un autre optimum local.

V.4 Mesure de comparaison entre les algorithmes

Selon les auteurs des algorithmes de l'optimisation continue, la comparaison entre ces algorithmes n'est pas basée sur le temps d'exécution mais sur **le nombre des évaluations de la fonction objectif** nécessaire pour atteindre la qualité de solution voulue [22].

D'après eux, cette méthode présente quelques avantages :

— résolution du problème des algorithmes implémentés en utilisant différents langages de programmation,
— insensibilité aux compétences du programmeur en termes d'optimisation du code,
— insensibilité à la performance du compilateur utilisé,
— facilité de comparaison des résultats obtenus par différentes machines.

Nous remarquons que cette méthode ne prend pas en considération la complexité de l'algorithme qui pourtant s'avère primordiale pour indiquer sa rapidité. En effet, la bonne mesure d'évaluation doit tenir compte de l'effectivité et de l'efficience de la méthode et exprimer le rapport entre ces deux critères.

Nous avons adopté cette méthode pour comparer CBSO aux autres algorithmes d'optimisation continue, mais nous présentons quand même le temps d'exécution de CBSO pour chaque fonction de test afin de permettre aux chercheurs de comparer leurs résultats avec les nôtres.

DP	**SP**	$\mathbf{B_2}$	**RC**	**ES**	**DJ**	$\mathbf{R_2}$	$\mathbf{H_{3,4}}$	$\mathbf{H_{6,4}}$	$\mathbf{S_{4,5}}$	$\mathbf{S_{4,7}}$	$\mathbf{S_{4,10}}$	$\mathbf{Z_2}$	$\mathbf{Z_5}$
205	19	1794	1	15	2	10	10	262	416	400	351	1	8

Tableau V.2. Temps d'exécution de CBSO pour les fonctions en millisecondes

Malgré l'espace de recherche infini, nous remarquons que CBSO est très rapide bien que les méta-heuristiques sont en général un peu lentes.

Nous avons comparé CBSO avec plusieurs algorithmes d'optimisation continue :

— ceux basés sur l'apprentissage des probabilités qui modélisent et échantillonnent explicitement des distributions des probabilités (CSA-ES, (1+1)ES, CMA-ES, IDEA et MBOA),

— ceux qui s'inspirent du comportement naturel des fourmis ($ACO_{\mathbb{R}}$, $ACO_{\mathbb{R}}$-PT, CACO et CIAC),

— ceux qui ont été initialement développés pour les problèmes combinatoires et qui ont été adaptés par la suite aux problèmes continus (CGA, ECTS, ESA, TS, CTS, INTEROPT, $CRTS_{min}$ et $CRTS_{ave}$).

Pour assurer une comparaison juste, nous répliquons le même environnement de test utilisé par les autres algorithmes, en particulier le domaine de définition des fonctions et la précision souhaitée.

V.5 Évaluation de CBSO

Pour une meilleure précision, nous avons mené 1000 exécutions indépendantes pour chaque fonction de test et nous avons considéré la moyenne du nombre des évaluations de chaque fonction.

Le meilleur algorithme d'optimisation continue est celui qui présente le plus petit nombre d'évaluations des fonctions. Les résultats d'un tel algorithme sont donnés en gras dans les tableaux suivants.

Le taux des exécutions réussies (où l'algorithme n'est pas stagné dans un optimum local) est donné entre crochets pour les fonctions multi-modales. Lorsque ce taux n'est pas précisé, il est égal à 100%.

	CBSO	**(1+1)ES**	**CSA-ES**	**CMA-ES**	**IDEA**	**MBOA**
DP	**398**	833	1258	1088	∞	40970
SP	**886**	1370	2192	1781	6850	65760

Tableau V.3. Premier tableau comparatif

	RC	**R_2**	**$H_{3,4}$**	**$H_{6,4}$**	**$S_{4,5}$**	**$S_{4,7}$**	**$S_{4,10}$**	**Z_2**	**Z_5**
CBSO	**112**	**471**	**578**	**418** [58%]	**2309** [19%]	**2671** [17%]	**2469** [24%]	**93**	**559**
CTS	668	1616	628	2250	4889 [50%]	5110 [60%]	5232 [56%]	689	27659

Tableau V.4. Deuxième tableau comparatif

	CBSO	**$ACO_{\mathbb{R}}$**	**$ACO_{\mathbb{R}}$-PT**	**CGA**	**ECTS**	**ESA**	**$CRTS_{min}$**	**$CRTS_{ave}$**	**INTEROPT**
$S_{4,5}$	2309 [19%]	787 [57%]	437	610	854 [75%]	1159 [54%]	664	812	**370** [40%]
$S_{4,7}$	2671 [17%]	748 [70%]	**173**	680 [83%]	884 [80%]	1224 [54%]	871	960	2426 [60%]
$S_{4,10}$	2469 [24%]	715 [81%]	**172**	650 [81%]	910 [75%]	1170 [50%]	693	921	3463 [50%]

Tableau V.5. Troisième tableau comparatif

	CBSO	**$ACO_{\mathbb{R}}$**	**$ACO_{\mathbb{R}}$-PT**	**CGA**	**ECTS**	**CIAC**	**CACO**	**ESA**	**$CRTS_{min}$**	**$CRTS_{ave}$**	**TS**	**INTEROPT**
B_2	431 [44%]	544	-	**430**	-	11968	-	-	-	-	-	-
RC	112	857	210	612	245	-	-	-	41	**38**	492	4172
ES	**94**	772	-	1466	-	-	-	-	-	-	-	-
DJ	**130**	397	-	744	-	-	-	-	-	-	-	-
R_2	**471**	820	2319	960	480	11480	6808	816	-	-	-	-
$H_{3,4}$	578	342	**55**	581	547	-	-	684	609	513	508	1113
$H_{6,4}$	**418** [61%]	722	1032	9386	1516	-	-	2671	1245	750	2845	17262
Z_2	93	292	**12**	624	195	50040	-	15795	-	-	-	-
Z_5	559	**727**	-	1381	2253	-	-	69792	-	-	-	-

Tableau V.6. Quatrième tableau comparatif

V.6 Analyse et discussion des résultats

L'important, après avoir présenté un algorithme d'optimisation, est de pouvoir évaluer sa performance et de le comparer avec d'autres algorithmes conçus pour le même objectif.

Selon Wolpert et Macready 1997 [31], aucun algorithme d'optimisation n'est plus adapté que les autres pour résoudre tous les types de problèmes. Mais cela n'empêche pas certains algorithmes d'être mieux adaptés que d'autres pour certaines classes de problèmes.

— Premier tableau comparatif (V.3) : Nous remarquons que pour les fonctions Diagonal Plane (DP) et Sphere (SP), CBSO est bien meilleur que les algorithmes (1+1)ES, CSA-ES, CMA-ES, IDEA et MBOA. En effet, ces algorithmes nécessitent plus d'évaluation des fonctions pour atteindre l'optimum global.

— Deuxième tableau comparatif (V.4) : Nous remarquons que CBSO donne de meilleurs résultats par rapport à CTS pour toutes les fonctions de test, tandis que ce dernier présente plus d'exécutions réussies lorsqu'il s'agit des fonctions multi-modales.

— Troisième tableau comparatif (V.5) : Malheureusement, pour la fonction Shekel à 5, 7 et 10 variables, CBSO est moins performant que les autres algorithmes d'optimisation.

— Quatrième tableau comparatif (V.6) : CBSO est plus performant que les autres algorithmes pour 4/9 fonctions de test. Pour le reste des fonctions, les résultats que donne CBSO ne sont pas très éloignés de ceux du meilleur algorithme.

Conclusion

Nous avons présenté les fonctions de test utilisées ainsi que les paramètres de notre algorithme. Ensuite, nous avons analysé le comportement de ce dernier face aux fonctions uni-modales et multi-modales, et à partir des résultats obtenus, nous avons fait une étude comparative entre CBSO et d'autres algorithmes d'optimisation continue.

Conclusion générale

Les machines dotées d'un système performant ont facilité le traitement et le stockage des données mais cela n'est pas suffisant parce que nous devons penser à des solutions optimales indépendantes des capacités de la machine. D'où l'intérêt de l'intelligence artificielle qui accélère le rythme des découvertes dans plusieurs domaines parmi lesquels la médecine, la chimie, l'agriculture, le commerce et l'éducation.

Les problèmes d'optimisation globale sont, depuis les deux décennies passées, un des domaines de recherche les plus compliqués .

Dans ce projet, nous avons pu montrer comment la méta-heuristique BSO, qui a été initialement conçue pour des problèmes d'optimisation combinatoire, peut être adaptée aux problèmes d'optimisation continue sans apporter beaucoup de modifications dans sa structure originale.

Les expérimentations ont montré l'efficacité de notre méta-heuristique CBSO pour les problèmes d'optimisation continue tels que *Diagonal Plane, Sphere, De Joung, Easom, B2, Rosenbrock, Zakharov, Hartmann* et *Branin RCOS*.

Une amélioration possible de notre algorithme serait de prendre en charge les problèmes à variables mixtes, c'est-à-dire pouvant être discrètes ou continues.

Une autre amélioration serait d'utiliser une heuristique pour restreindre les domaines de définition de la fonction problème au fur et à mesure des itérations pour augmenter le nombre de chances pour trouver l'optimum en réduisant la taille de l'espace de recherche.

Nous pourrions encore relever le défi et passer au cas de l'optimisation multi-objectif pour satisfaire plus d'un critère en même temps.

De nos jours, l'optimisation reste un domaine de recherche fécond et un sujet de plusieurs travaux de la recherche opérationnelle car elle s'applique dans tous les domaines de l'activité humaine prêtant à la modélisation mathématique.

Bibliographie

[1] Babazadeha Abbas, Poorzahedy Hossain et Nikoosokhan Saeid: *Application of particle swarm optimization to transportation network design problem.* Journal of King Saud University-Science, 23(3) :293–300, 2011.

[2] Battiti Roberto et Tecchiolli Giampietro: *The continuous reactive tabu search : blending combinatorial optimization and stochastic search for global optimization.* Annals of Operations Research, 63(2) :151–188, 1996.

[3] Bilbro Griff L. et Snyder Wesley E.: *Optimization of functions with many minima.* IEEE Transactions on Systems, Man, and Cybernetics, 21(4) :840–849, 1991.

[4] Bilchev George et Parmee Ian: *The ant colony metaphor for searching continuous design spaces.* Evolutionary Computing, pages 25–39, 1995.

[5] Bosman Peter et Thierens Dirk: *Multi-objective optimization with diversity preserving mixture-based iterated density estimation evolutionary algorithms.* International Journal of Approximate Reasoning, 31(3) :259–289, 2002.

[6] Boussaid Ilhem: *Improvement of metaheuristics for continuous optimization.* Thèse de doctorat, Université Paris-Est, 2013.

[7] Chelouah Rachid et Siarry Patrick: *A Continuous Genetic Algorithm Designed for the Global Optimization of Multimodal Functions.* Journal of Heuristics, 6 :191—-213, 2000.

[8] Chelouah Rachid et Siarry Patrick: *Tabu Search applied to global optimization.* European Journal of Operational Research, 123 :256—-270, 2000.

[9] Cvijovic Djurdje et Klinowski Jacek: *Taboo search : an approach to the multiple minima problem.* Science, 267(5198) :664, 1995.

[10] Djeffal Marwa et Drias Habiba: *Multilevel Bee Swarm Optimization for Large Satisfibility Problem Instances.* Dans *Intelligent Data Engineering and Automated Learning*, tome 8206, pages 594–602, 2013.

[11] Djenouri Y., Drias H. et Habbas Z.: *Bee Swarm Optimization using multiple strategies for association rules mining.* Dans *International Journal of Bio-Inspired Computation*, tome 6, pages 239–249, 2014.

[12] Dorigo Marco, Birattari Mauro et Stutzle Thomas: *Ant colony optimization.* IEEE computational intelligence magazine, 1(4) :28–39, 2006.

[13] Dorigo Marco, Maniezzo Vittorio et Colorni Alberto: *Ant system : optimization by a colony of cooperating agents.* IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 26(1) :29–41, 1996.

[14] Dréo Johann et Siarry Patrick: *Continuous interacting ant colony algorithm based on dense heterarchy.* Future Generation Computer Systems, 20(5) :841–856, 2004.

[15] Drias Habiba, Douib Ameur et Hireche Célia: *Swarm Intelligence with Clustering for Solving SAT.* Dans *Intelligent Data Engineering and Automated Learning*, tome 8206, pages 585–593, 2013.

[16] Drias Habiba, Sadeg Souhila et Yahi Safa: *Cooperative Bees Swarm for Solving the Maximum Weighted Satisfiability Problem.* Dans *Computational Intelligence and Bioinspired Systems*, tome 3512, pages 318–325, 2005.

[17] Esmin Ahmed A. et Lambert-Torres Germano: *Application of particle swarm optimization to optimal power systems.* International Journal of Innovative Computing, Information and Control, 8(3A) :1705–1716, 2012.

[18] Hansen Nikolaus, Müller Sibylle D. et Koumoutsakos Petros: *Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES).* Evolutionary computation, 11(1) :1–18, 2003.

[19] Jean-Louis Rouget: *Notes de cours.* 2012.

[20] Kennedy James et Eberhart Russell: *Particle Swarm Optimization.* Dans *Proceedings of IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[21] Kern Stefan, Müller Sibylle D., Hansen Nikolaus, Büche Dirk, Ocenasek Jiri et Koumoutsakos Petros: *Learning probability distributions in continuous evolutionary algorithms–a comparative review.* Natural Computing, 3(1) :77–112, 2004.

[22] Krzysztof Socha et Marco Dorigo: *Ant colony optimization for continuous domains.* Dans *European Journal of Operational Research*, tome 185, pages 1155–1173, 2006.

[23] Ma Rong-Jiang, Yu Nan-Yang et Hu Jun-Yi: *Application of particle swarm optimization algorithm in the heating system planning problem.* The Scientific World Journal, 2013.

[24] Momin Jamil et Yang Xin-She: *A literature survey of benchmark functions for global optimization problems.* Journal of Mathematical Modelling and Numerical Optimisation, 4(2) :150—-194, 2013.

[25] N. Hu: *Tabu Search method with random moves for globally optimal design.* International Journal for Numerical Methods in Engineering, 35 :1055–1070, 1992.

[26] Ocenasek Jiff et Schwarz Josef: *Estimation of distribution algorithm for mixed continuous-discrete optimization problems.* Dans *2nd Euro-International Symposium on Computational Intelligence*, pages 227–232, 2002.

[27] Riadi Indra Chandra Joseph *et al.*: *Cognitive Ant colony optimization : A new framework in swarm intelligence*. Thèse de doctorat, University of Salford, 2014.

[28] Seeley Thomas D., Camazine Scott et Sneyd James: *Collective decision-making in honey bees : how colonies choose among nectar sources*. Behavioral Ecology and Sociobiology, 28(4) :277–290, 1991.

[29] Siarry Patrick et Berthiau Gérard: *Fitting of tabu search to optimize functions of continuous variables*. International journal for numerical methods in engineering, 40(13) :2449–2457, 1997.

[30] Siarry Patrick, Berthiau Gérard, Durdin François et Haussy Jacques: *Enhanced simulated annealing for globally minimizing functions of many-continuous variables*. ACM Transactions on Mathematical Software (TOMS), 23(2) :209–228, 1997.

[31] Wolpert David H. et Macready William G.: *No free lunch theorems for optimization*. IEEE transactions on evolutionary computation, 1(1) :67–82, 1997.

[32] Yang Xin-She: *A new metaheuristic bat-inspired algorithm*. Nature inspired cooperative strategies for optimization (NICSO 2010), pages 65–74, 2010.

Résumé

Les méta-heuristiques sont des algorithmes génériques souvent bio-inspirés, qui servent à résoudre des problèmes complexes. Bee Swarm Optimization (BSO) est une méta-heuristique prouvée efficace pour résoudre certains problèmes d'optimisation combinatoire.

Le but de ce travail est d'adapter BSO à l'optimisation des fonctions à variables continues. Nous l'avons donc fait par la redéfinition de quelques principes de base tels que le voisinage d'une solution.

L'efficacité de notre algorithme que nous avons appelé Continuous Bee Swarm Optimization (CBSO) a été prouvée à travers un ensemble de fonctions multi-modales dont les optimums globaux sont connus.

Mots-clefs : Optimisation globale, optimisation difficile, variables continues, intelligence artificielle, méta-heuristiques, Bee Swarm Optimization.

Abstract

Metaheuristics are generic algorithms, often bio-inspired, which aim to solve complex problems. Bee Swarm Optimization (BSO) is a metaheuristic proved efficient to handle some combinatorial optimization problems.

The goal of this work is to adapt BSO to optimize functions with continuous variables. Hence, we have done it by redefining some basic concepts such as solution neighbourhood.

The efficiency of our algorithm called Continuous Bee Swarm Optimization (CBSO) has been proved through a set of multi-modal functions for which minima are known.

Keywords : Global optimization, hard optimization, continuous variables, artificial intelligence, metaheuristics, Bee Swarm Optimization.