Conception d'un module de tests de métaheuristiques

Rapport de stage

Maîtrise d'informatique

Université de Cergy-Pontoise

Jean-Philippe Aumasson

Sous la direction de Johann Dréo

LISSI, Université Paris 12 Val-de-Marne
61, avenue du Général de Gaulle 94010, Créteil cedex
tél : 01.45.17.14.94
fax : 01.45.17.14.92

Avant-propos

J'ai dès le début de l'année voulu effectuer mon stage de fin de diplôme dans un centre de recherche, et j'ai ainsi pris contact avec Johann Dréo fin octobre 2004, une offre de stage type bac +5 étant proposée sur sa page personnelle. Après un entretien quelques jours après nous nous sommes entendus pour un stage d'une durée de deux mois, relatif à l'étude automatisée des métaheuristiques, domaine quelque peu marginal de l'optimisation. Nous reviendrons plus en détails sur le sujet dans la deuxième partie du rapport. A l'époque le laboratoire se nommait encore LERISS, aujourd'hui c'est le LISSI, regroupant plusieurs activités, qui m'accueillera. Mon travail sera dans la continuité du projet Open Metaheuristic, initié par Johann Dréo en début d'année, dont le code source est disponible publiquement sur le site de l'hébergeur, à l'adresse suivante : *http ://ometah.berlios.de.*

Comme c'est l'usage, quelques lignes seront accordées à la présentation du laboratoire, nous livrerons ensuite une introduction générale au stage présentant les notions théoriques, et la mission qui me sera confiée. La seconde partie décrivant mon activité au cours de ces deux mois est divisée en deux chapitres : le premier concernera la partie la plus importante de mon travail, soit la conception du module de tests, le second présentera quelques algorithmes implémentés, existants ou originaux, avec des comparaisons de leurs performances. Après avoir conclu, on pourra trouver en annexe les spécifications (formule et graphe) des problèmes utilisés qui n'ont pas été détaillés dans le corps du rapport, et enfin de courtes portions du code produit.

Table des matières

Première partie

Introduction

Chapitre 1

Le laboratoire LISSI

Le laboratoire m'accueillant est donc le LISSI (Laboratoire Images, Signaux et Systèmes Intelligents, EA 3956, dirigé par Patrick Siarry) , nouvellement crée début 2005, résultat de la fusion de trois laboratoires de l'Université Paris 12, le LERISS (EA 412), le LIIA (EA 1613), et le I2S (JE 2353), comptant au total soixante-deux chercheurs : six professeurs, vingt maîtres de conférences, vingt-huit doctorants, six chercheurs post-doc et deux professeurs associés. Les activités sont dispersés sur trois sites : Créteil, Vitry-sur-Seine, et Melun-Sénart. Nous présenterons brièvement les activités des différentes équipes du labo, en résumant la présentation fournie sur le site Internet :

- Traitement de l'image et du signal :
  - Traitement du Signal :
    - Optimisation : extraction du signal du bruit, métaheuristiques pour les problèmes d'optimisation difficile.
    - Analyse multi-échelle :caractérisation des signaux physiologiques (potentiels évoqués auditifs ou otholitiques, signaux posturographiques, electrocardiogrammes).
    - Compression : compression d'électrocardiogrammes avec préservation de l'information clinique, méthodes d'analyse et de débruitage.
  - Traitement d'image :
    - Segmentation d'image : techniques à base de programmation dynamique, calcul variationnel, propagation de fronts.
    - Applications : analyse des angiographies de fond d'oil, caractérisation de l'ischémie myocardique (images IRM), étude phénoménologique comparative de la brûlure thermique et du syndrome cutané d'irradiation aiguë par imagerie optique des milieux diffusants.
- Systèmes Complexes, Traitement de l'Information et de la Connaissance :
  - Modélisation et simulation :
    - Modélisation dynamique de structures polyarticulées ouvertes ou fermées permettant une parallélisation des calculs. Réduction de complexité al-

gorithmique.
  - Modélisation et caractérisation des robots redondants continuum. Optimisation des architectures de robots.
  - Modélisation de l'interaction système-environnement. Algorithmes de détection de collisions et de synthèse de retour d'efforts.
- Traitement intelligent de l'information et de la connaissance - Approches bio-inspirées :
  - Méthodes de classification floue. Génération et optimisation par apprentissage de systèmes à inférence floue.
  - Modèles neuronaux, structures neuronales hybrides multi-modèles et multi-réseaux de neurones.
  - Méthodes de contrôle et d'optimisation des systèmes distribués : Prise en compte de contraintes temps réel et de qualité dans le transport des flux d'informations.
  - Systèmes auto-organisés et coopérants au sens de l'Intelligence Artificielle Distribuée.

C'est au sein de l'équipe Optimisation que j'effectuerai mon stage. Mon maître de stage a soutenu sa thèse l'an passé, en spécialité génie biologique et médical, intitulée "Adaptation de la méthode des colonies de fourmis pour l'optimisation en variables continues. Application en génie biomédical", et actuellement d'autres thèsards de l'équipe optimisation étudient les méthodes d'optimisation par essaims de particules, et par colonies d'insectes notamment, et des membres du laboratoires se sont auparavant interessés à d'autres aspects de l'optimisation, notamment multiobjectif. On retrouvera plus d'informations sur le nouveau site du laboratoire : *http://www.univ-paris12.fr/13569540/0/fiche___pagelibre/*.

Chapitre 2

Sujet d'étude

2.1 Optimisation discrète et continue

La notion d'optimisation n'est pas récente, les commerçants n'ont pas attendu la programmation linéaire pour essayer de calculer les paramètres optimisant leur gain financier, mais la formalisation mathématique et la création de méthodes de recherche complexes a bien moins d'un siècle. On définit généralement une situation d'optimisation comme le problème consistant à trouver les paramètres à donner à une fonction telle que la valeur retournée par celle-ci soit optimale (minimale ou maximale). On nomme alors fonction objectif, ou fonction d'évaluation, la fonction en question. Des contraintes peuvent être imposées pour la recherche de la meilleure solution (espace de recherche, etc.). On peut diviser l'optimisation en deux domaines, selon que le problème soit discret (optimisation combinatoire) ou continu. On pourra aussi trouver des problèmes mixtes, où certaines variables seront discrètes, d'autres continues. La fonction d'évaluation est souvent associée à un problème difficile, dont la solution ne pourraît être trouvée systématiquement avec un algorithme donné en temps raisonnable (problèmes NP-difficiles). Il existe quantité de méthodes pour "résoudre" ces problèmes, en fonction du besoin (précision, rapidité de calcul, etc.), citons par exemple les méthodes d'approximation réduisant la complexité du problème, la programmation linéaire, ou non-linéaire, les heuristiques spécialisées, ou encore les métaheuristiques, qui sont des méthodes approchées de recherche globale.

2.2 Métaheuristiques

Les premières métaheuristiques datent des années 80, et bien que d'origine discrètes, on peut les adapter à des problèmes continus. Elles interviennent généralement quand les méthodes classiques ont échoué, et sont d'une efficacité

relativement imprévisible. Le terme métaheuristique est utilisé car, par opposition aux heuristiques particulières pour un problème donné, les métaheuristiques peuvent être utilisées pour plusieurs types de problèmes, et l'heuristique n'est pas réellement explicite, mais déduite d'un algorithme donné. Les métaheuristiques ont également comme caractèristiques communes leur caractère plus ou moins stochastique, ainsi que leur inspiration par une analogie avec d'autres sciences (physique, biologie, etc.).

On ne peut pas vraiment citer un domaine d'application de prédilection, on pourra retrouver un problème d'optimisation dans de nombreuses situations, l'important pour nous n'étant pas la signification des symboles utilisés (l'application concrètre du problème) mais plutôt la complexité de leurs relations. On pourra par exemple livrer une optimisation pour un problème de trafic aérien, de télécommunication (routage), ou pour un classique problème de tournée, et on constatera parfois que deux problèmes sont presque équivalents formellement alors qu'ils ont des domaines d'application totalement étrangers. De toute façons, si deux problèmes difficiles sont NP-complet, on pourra théoriquement appliquer la résolution de l'un à l'autre dans un temps raisonnable.

Les métaheuristiques ne sont pas des méthodes figées, il n'y a pas de relation d'ordre quant à l'efficacité absolue d'un algorithme ou d'un autre, tout dépend des paramètres utilisés, du problème d'application, d'où l'interêt de simulations informatisées.

2.3 Problèmes

2.3.1 Présentation

Comme on l'a dit, une métaheuristique ne sera pas forcément absolument meilleure que telle ou telle autre, et il est indispensable de tester ces méthodes d'optimisation sur des problèmes "réels". Les problèmes utilisés pour les tests et benchmarks d'algorithmes sont apparentés à des fonctions de l'espace vectoriel des solutions dans celui des valeurs, rationnelles. On optimisera (on minimisera généralement, une maximisation revenant à minimiser l'inverse) la valeur de la fonction d'évaluation associée au coordonnées des solutions potentielles, qu'on nommera points. La difficulté de l'optimisation dépendra de l'allure de la fonction (irrégularités, discontinuités, etc.), et pour les tests on supposera connu l'emplacement de l' (des) optimum (optima). Pour éviter toute confusion, précisons qu'on emploiera en général le terme optima comme pluriel d'optimum (optimums et optima sont tous deux tolérés), et qu'on tâchera d'optimiser. On appliquera cette règle à maximum, minimum et extremum.

Les représentations dans un espace à trois dimensions donnent une allure de la fonction et de sa complexité, mais si l'instance du problème dispose de plus de trois dimensions, les figures en deux dimensions ne peuvent pas décrire complètement le problème, et ne sont qu'une coupe de l'espace à N dimensions en un

espace de dimension moindre. Pour certaines fonctions "simples", comme Sphere qui est d'allure parabollique, on pourra s'imaginer dans un espace à N dimensions le creux dont le fond représente le minimum, la coordonnée de chaque dimension s'accroissant au fur et à mesure que l'on s'éloigne de l'optimum (au sens euclidien). Mais on appréhendera plus difficilement une fonction au comportement moins prévisible, certaines fonctions présentant même des discontinuités. La symétrie, la convexité, et la quantité d'optima locaux du problème seront des critères déterminantes de la difficulté d'approximation de l'optimum réel.

Dans les problèmes considérés, on cherchera en général à minimiser la valeur de la fonction d'évaluation, dont on connaîtra l'optimum réel, ce qui sera notamment utile pour afficher l'évolution de l'erreur relative des évaluations effectuées. On pourra s'étonner en testant le programme que l'arbre XML soit conçu de telle façon qu'on puisse afficher plusieurs optima, mais on doit envisager le cas où plusieurs points de l'espace vectoriel associé au problème aient la même évaluation minimale. Les fonctions admettant plusieurs minima (locaux ou globaux) sont appellées multimodales, celles n'ayant qu'un optimum solution sont dites unimodales. Chaque problème étant décrit par une fonction d'évaluation, on parlera alors de problèmes unimodaux ou multimodaux. Comme exemples triviaux de fonctions unimodales et multimodales, on peut respectivement citer les fonctions paraboliques (non-inversées) et les fonctions sinuoïdales, à un nombre quelconque de dimensions. On pourra de plus admettre une certaine tolérance sur les valeurs, pour considérer comme optimaux deux points aux évaluations très proches, dont la plus basse constituera le minimum "réel", et sera considéré pour les calculs de marge d'erreur.

Dans la limite du possible, on utilisera des fonctions décalées (*shifted*), de telle façon que l'optimum se trouve à l'origine du repère (0 ... 0), où ayant des coordonnées naturelles (1..1). Pour celà le repère est décalé dans chaque dimension de la coordonnée de l'optimum sur cette dimension. On devra vérifier que l'espace des solution après ce décalage se trouve bien dans une zone non-triviale, et ne masque pas la difficulté du problème.

Une des difficultés majeures dans tout problème d'optimisation est la présence d'optima locaux, soit de points possédant une valeur minimale sur un voisinage raisonnable par rapport à l'espace de recherche considéré. On peut illustrer cette notion par des cavités de profondeur variable, dont le plus profond constituera l'optimum réel. La difficulté a priori dépend donc de la complexité de la fonction, de ses irrégularités, et éventuellement discontinuités, et en pratique une plus grande difficulté aura pour conséquence une moindre précision pour une recherche équivalente, ou réciproquement un plus long calcul pour obtenir une même précision.

La plupart des problèmes implémentés dans le logiciel Open Metaheuristic sont issus de [6], certains étant centrés pour avoir la valeur de l'optimum en une extremité de l'espace de recherche.

2.3.2 Exemples

On présentera quelques fonctions de benchmark courantes pour les problèmes d'optimisation, qu'on retrouvera notamment dans les tests de performances (voir 4.6). Dans chacune des fonction $F_i$présentées, x est un point de l'espace vectoriel des solutions, les $x_i, i = 1..D$ étant les coordonnées de x dans la base considérée pour chaque dimension.

Exemples de problèmes unimodaux

Schwefel

Fonction relativement simple, les contours autour de l'optimum sont des ellipses, aucun minimum local autre que le global n'est présent, on trouvera facilement une solution correcte, et une haute précision sera demandée (10^{-6}).

$F_1(x) = \sum_{i=1}^{D}(\sum_{j=1}^{i}(x_j^2))$

Sphere

C'est une des fonctions les plus simples, la cavité du minimum est explicite, la représentation est d'allure parabollique, on exigera une grande précision par rapport à la valeur (10^{-6}) pour considérer un point comme solution acceptable.

$F_2(x) = \sum_{i=1}^{D} x_i^2$

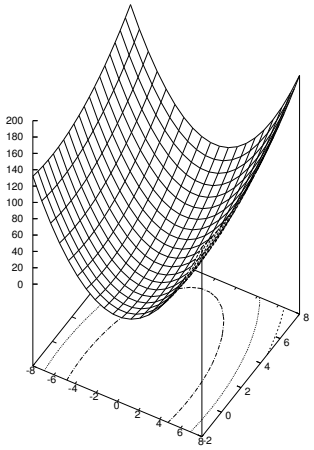
Exemples de problèmes multimodaux

Rastrigin Une fonction trigonométrique en "champ de bosses", la difficulté est alors accrue par le grand nombre de minima locaux.

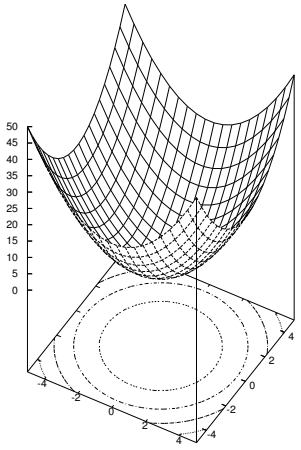
$F_3(x) = \sum_{i=1}^{D}(x_i^2 - 10\cos(2\pi x_i) + 10)$

Rosenbrock Cette fonction n'a pas à subir de décalage, son optimum se trouve aux coordonnées $1, ..., 1$ dans l'espace des solutions considérés, et se trouve dans une "vallée" étroite comportant plusieurs minima locaux, rendant difficile la recherche.

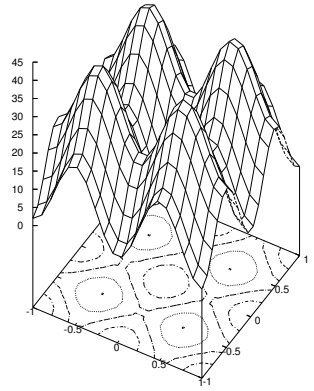
$F_4(x) = \sum_{i=1}^{D-1}(100 \cdot (x - x_{i+1})^2 + (x_i - 1)^2)$



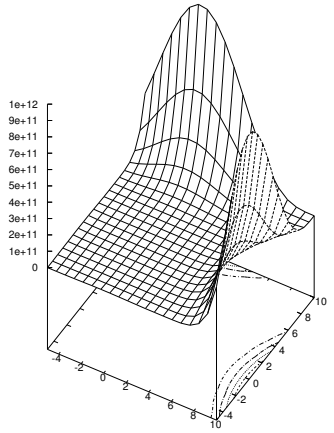
(a) Fonction Schwefel



(b) Fonction Sphere



(c) Fonction Rastrigin.



(d) Fonction Rosenbrock.

FIG. 2.1 – Représentations des problèmes en trois dimensions.

2.4 But du stage

La tâche qui m'a été confiée se trouvait en grande partie conditionnée par l'avancement du projet Open Metaheuristic, et lors de l'entretien en novembre la situation m'a été exposée, nous avons alors convenu que je me chargerais de développer un module de tests, et éventuellement de participer à Open Metaheuristic si sa programmation n'était pas suffisament avancée à mon arrivée. Le projet ayant été officiellement crée en début d'année 2005, publiquement consultable sur le site de l'hébergeur Savannah, j'ai pu suivre son développement, ma préparation au stage étant complétée par l'étude du livre dont mon maître de stage est co-auteur ([3]). L'essentiel de mon travail, les tests automatisés de métaheuristiques, consiste plus précisement en la conception d'un ensemble de tests statistiques et la sélection de critères judicieux, sous la forme d'un ensemble de modules en langage Python permettant la création de scripts. Je participerai sporadiquement, dans une moindre mesure et en notamment en fonction des besoins de mon sous-projet, à l'implémentation du logiciel principal, notamment pour la mise en places d'algorithmes

2.5 Outils

Les logiciels ou sites mentionnés dans ce rapport n'étant pas forcément connus de tout le monde (j'ai l'illusion que ce rapport soit lu par plus de trois personnes), présentons brièvements quelques-un des termes utilisés :

BerliOS Portail open source offrant gracieusement l'hébergement aux projets open source, et pas seulement sous licence libre ((L)GPL...), qui héberge actuellement le projet Open Metaheuristic. *http ://www.berlios.de*

C++ Le célèbre langage compilé utilisé pour le module principal ometah, orienté-objet. Le compilateur utilisé est gcc (g++) version 3 (3.3 ou 3.4), mais des erreurs à la compilation ont été signalées avec le récent gcc 4, pour d'obscurs problèmes de templates.

CVS Logiciel pour la gestion de sauvegarde de projets, et d'accès concurrents par les différents développeurs (*Concurrent Versions System*), on l'utilisera pour sauver le projet sur le serveur de l'hébergeur. On parlera de *commit* pour appliquer les modifications réalisées sur les fichiers du serveurs.

Python Langage interprété de haut niveau, orienté objet, non explicitement typé, adapté pour le traitement de données de gros programmes (C par exemple), disposant de nombreux modules en open source. La définition de blocs par indentation a pour conséquence une bonne lisibilité du code (meilleure que Perl par exemple), et la consision de sa syntaxe nécessite beaucoup moins de ligne de codes qu'un programme C par exemple, pour réaliser la même tâche. *http ://www.python.org*

R Logiciel de calcul statistique et de génération des graphiques associés, version libre de S, il s'avèrera bien plus complet qu'un gnuplot par exemple, même si il ne brille pas par la simplicité de son utilisation ni par la quantité de ressources systèmes nécessaires. *http ://www.r-project.org*

rpy Module Python créant une interface pour R, permettant ainsi de réaliser la plupart des fonctions de R dans un module Python, en créant une correspondance entre les structures et types de chaque langage. *http ://rpy.sourceforge.net*

Savannah Hébergeur de projets libres, satellite du projet GNU, qui a hébergé Open Metaheuristic les premiers mois, nous relaterons brièvement les raisons de notre rupture. *http ://savannah.gnu.org*

scons Le project builder utilisé pour le module C++ ometah, logiciel écrit en Python, aux fonctions plus ou moins analogues à celles d'un autoconf ou automake. *http ://www.scons.org*

Nous travaillons dans un environnement Linux, de ce fait la compatibilité sera assurée pour tout utilisateur sous Linux muni des logiciels requis, dans les versions adéquates (récentes en général), mais on veillera à assurer le fonctionnement du programme, du moins sa possibilité théorique, sous un système Windows. Tous les logiciels nécessaires y sont utilisables (portages, ou au pire, version émulées), et on n'utilisera pas de fonctions propres à Linux (appels systèmes, commandes bash, séparateurs, etc.).

2.6 Le projet Open Metaheuristic

2.6.1 Présentation

Open Metaheuristic est un projet visant à tester des métaheuristiques sur un ensemble de problèmes, programmé en C++ et donc d'implémentation fortement modulaire, on doit facilement pouvoir ajouter des fonctions de tests, algorithmes, pour en comparer les performances. Initié en début d'année, j'ai suivi son développement en consultant le dépôt CVS en ligne sur Savannah, les premières étapes consistant surtout en l'implémentation du modèle objet, et notamment le système de communication entre les différents objets. Le design pattern abstract factory est utilisé pour regrouper les différentes instances d'une même surclasse : métaheuristiques, problèmes, classes de communication.

Le projet sera divisé en deux modules ; le principal, ometah, le programme en C++ calculant l'optimisation, et le module de tests, ometahlab, dont la réalisation m'était chargée.

Sous licence libre, il pourra être librement utilisé par d'autres chercheurs à travers le monde pour effectuer des tests de métaheuristiques, et éventuellement participer à l'amélioration du logiciel, en signalant des bugs, ou en programmant des fonctions inédites. La portée potentiellement universelle du travail

réalisé nous contraindra, et c'est une bonne chose, à adopter une certaine rigueur ; propreté du code, de la terminologie employée, lisibilité des fonctions et intelligibilité du fonctionnement, en n'employant évidemment que des termes anglophones, que celà soit pour le nommage des entités du programme, les commentaire, ou même pour les fils de discussion publics.

La page du projet chez l'hébergeur berliOS : *http ://ometah.berlios.de*

2.6.2 Logiciel libre

Avant de développer les détails de mes travaux, mentionnons une petite anecdote survenue durant le développement du projet. Au départ hébergés par Savannah (*http ://savannah.gnu.org*), site accueillant de nombreux projets (libre, *sine qua non*) et supervisé par la communauté GNU (*http ://gnu.org*). On connaît les querelles de clocher entre les apôtres du code ouvert et les adeptes du logiciel *free as freedom*, cependant nous n'imaginions pas devoir en faire les frais, ayant rigoureusement publié les licences LGPL sur chacun des sources, et évidemment scrupuleusement respecté les quelques règles qu'elle impose. Mais plusieurs mois après la création du projet sur Savannah, un mail d'un superviseur de l'hébergeur libre nous informe que le nom du projet (Open Metaheuristic) doit être modifié, le "Open" n'ayant selon lui pas sa place, et il nous impose de le remplacer par un "Free" plus de circonstance étant donné la licence utilisée. Ce détail nous semblant futile, et l'assonance en "i" conséquente d'un renommage désagréable, nous décidons à l'unanimité (deux voix pour, zéro contre) de conserver le nom Open Metaheuristic. Après négociation avec le représentant du site, n'ayant pas trouvé de point d'accord, nous n'avons d'autre solution que la rupture avec l'hébergeur à cornes, et on se dirige rapidement vers un autre hôte de projets, l'allemand BerliOS (*http ://www.berlios.de*). Bien que soutenant l'idée du logiciel libre (j'ai réalisé plusieurs projets sous GPL), le glissement vers une sorte d'idéologie ou croyance mystique en la toute puissance de la GPL m'a toujours paru idiot, et cette anecdote est un peu révélatrice du sectarisme de certains (R.M. Stallman en tête), bien que je conçoive parfaitement que Savannah tienne à une certaine rigueur chez les projets qu'il héberge gracieusement, la séparation s'est d'ailleurs faite de façon très cordiale. Le fil du débat est publiquement consultable à l'adresse suivante : *https ://savannah.gnu.org/task/ ?func=detailitem&item_id=3620*

2.6.3 Conception de l'interface

Le premier travail a été de réaliser l'interface utilisateur du programme ometah : un problème et un algorithme (Rosenbrock et CEDA) étaient déjà implémentés, mais on ne pouvait lancer un calcul qu'en modifiant manuellement le code source du programme principal. L'objectif était de pouvoir lancer tout calcul en ligne de commande selon les paramètres mentionnés à l'aide des options, selon les conventions utilisées (flag court avec un tiret et une lettre, long avec deux

tirets et une chaîne plus longue). On gèrera également les options bloquantes courantes, affichant la version ou les informations d'usage du programme. Ces options bloquantes seront prioritaires sur les non-bloquantes, un ordre arbitraire étant fixé pour leur execution. L'interface réalisée n'offrera aucune interactivité, afin de permettre plus tard l'automatisation des calculs avec les modules de tests. Les erreurs sont détectées et l'utilisateur en est informé (option inconnue, valeur manquante ou superflue, etc.). Cette tâche est relativement simple en C++, le détail de l'implémentation apporterait ici peu d'interêt, on mentionnera seulement la création d'une classe *itsArgumentParser* dédiée au traitement de la ligne de commande, les arguments étant déclarés dans le programme principal, mentionnant les chaînes de caractère des flags, le type, la valeur par défaut, etc., de façon à pouvoir ajouter facilement de nouvelles options. Les trois premiers jours du stage ont donc été consacrés à la prise en main du programme, la compréhension de son fonctionnement (design patterns utilisés, nature des classes, etc.), et l'implémentation de l'interface décrite ci-dessus. L'affichage de l'usage est automatisé, on n'aura ainsi pas besoin de modifier une chaîne de caractère pour afficher un nouveau problème dans l'affichage.

Exemple de sortie

[jp@ometah]$ ./ometah -optioninconnue
Unknown option : -optioninconnue
Usage : ./ometah [options]
Options :
Short, Long Default Description
-v, –verbose (0) verbose level
-p, - -problem (Rosenbrock) problem name (Weierstrass,Rastrigin,Sphere,Ackley,Schwefel,Griewank,Rosenbrock)
-m, –metah (CEDA) metaheuristic name (SGEN,GS,JGEN,CEDA,RA,NMS,CHEDA)
-C, –comclient (Embedded) protocol name for client (Embedded)
-S, –comserver (Embedded) protocol name for server (Embedded)
-r, –random-seed (0) seed of the pseudorandom generator
-D, –debug () debug key
-i, –iterations (10) maximum number of iterations
-e, –evaluations (1000) maximum number of evaluations
-P, –precision (0.0) minimum value to reach
-s, –sample-size (10) number of points in the sample
-d, –dimension (1) dimension of the problem
-o, –output () output of the results
-V, –version check version
-h, –help show help
-u, –usage show usage informations

Deuxième partie

Activités

Chapitre 3

Le module de test

3.1 Introduction

3.1.1 Objectifs

La réalisation d'un module de tests de ometah constitue le principal objectif du stage : ma mission est d'implémenter en Python un système permettant l'étude des données en sorties d'ometah, soit l'automatisation de lancement répetés d'ometah selon des paramètres donnés, traiter les données recueillies et établir des critères judicieux pour étudier et comparer des calculs avec plusieurs ensembles de paramètres différents. Je dois également réaliser des graphiques portant sur divers critères statistiques. Initialement intégré à ometah dans le sous-répertoire experiment, on crée un module parallèle à ometah étant donné les divergences entre les deux programmes. En résumé, les informations en entrées seront des points dans l'espace de recherche, en sortie des graphiques et statistiques portant sur l'ensemble de points.

3.1.2 Projets existants

Comme la plupart des projets, la création d'Open Metaheuristic part d'un besoin, des programmes équivalents n'existant pas, du moins sous forme publique. Des implémentations d'algorithmes ont évidemment été réalisées, mais sont souvent spécifique à un type d'algorithme (évolutionnaire, essaims de particules, etc.), et/ou ne sont pas rendues publiques. De plus les implémentations sont souvent des recherches "brutes", où on ne trouve pas ou peu de statistiques sur l'évolution du calcul, seulement le résultat final et quelques statistiques de base, sur lesquelles le post-traitement s'effectuera à part, pour éventuellement générer des graphiques à l'aide de logiciels annexes. Notre projet permettra de lancer simplement des tests fortement paramètrables à l'aide de scripts Python, et d'en

générer les statistiques et graphiques automatiquement. Cette polyvalence aura aussi ses "mauvais côtés" : l'utilisation de C++, Python, R, et des librairies associées impliquera un grand nombre de pré-requis logiciels. Mais ceci ne constitue pas réellement un frein, dans la mesure où le logiciel n'est pas destiné au grand public, mais seulement à quelques spécialistes souvent habitués à ces outils, et dont l'installation éventuelle ne devrait pas poser de gros problèmes techniques.

3.1.3 Critères de test

Le prélude à la programmation de test est l'analyse des besoins, autrement dit l'identification des principaux critères d'évaluation d'une méthode d'optimisation. Le document [6] en fournit quelques uns, utilisés pour les benchmarks des programmes de la conférence. Mais avant d'expliciter ces critères, détaillons brièvement la méthode de génération des données : les métaheuristiques étant par définition stochastiques, on n'obtiendra pas à chaque lancement le même résultat. Ainsi pour chaque ensemble de paramètres, on lancement le calcul un certain nombre de fois, les runs, desquels nous extrairont les données soumises à l'étude statistique. Chaque lancement (on parlera désormais de runs) consiste en un ensemble d'itérations, le calcul s'arrêtant une fois le nombre maximal d'évaluations atteint, ou si l'optimum réel (connu) a suffisament été approché (on parlera de précision, ou marge d'erreur). On a donc deux critères d'arrêts possibles. Pendant le calcul les données modifiées sont les points d'un échantillon, chaque point étant muni d'une solution (vecteur de ses coordonnées dans l'espace de recherche), et d'une valeur (vecteur de valeurs pour chaque dimension de l'espace des valeurs, on se limitera dans notre étude à une dimension unique, l'optimisation d'un nombre supérieur nous amènerait dans le domaine de l'optimisation multiobjectif, plus complexe). On évaluera donc les algorithmes, en les comparant sur une même base (problèmes, précision, nombre d'évaluation), en retenant les critères suivants :

- Optimum trouvé (meilleure valeur pour un point).
- Rapidité de convergence des valeurs (évolution temporelle de l'échantillon).
- Distribution des valeurs des points de l'échantillon dans l'espace des valeurs.
- Evolution de l'erreur par rapport à la valeur optimale.
- Taux de réussite sur l'ensemble des runs.
- Positions dans l'espace des solutions.

Pour une étude rigoureuse, on devra effectuer les tests sur un grand nombre d'évaluations pour chaque run, sur des problèmes complexes, avec un espace des solutions de dimension allant jusqu'à cinquante. On pourra atteindre un nombre maximal de cinq-cent mille évaluations pour des problèmes à cinquante dimensions.

3.1.4 Contraintes

Le "cahier des charges", en plus des besoins scientifiques, impose quelques contraintes d'ordre techniques, le plus souvent peu problématiques, et réglés par une relative maîtrise des outils employés.

3.1.4.1 Utilisation en script

Le programme devra être divisé en modules, comme c'est l'usage dans la commauté Python, et permettre une utilisation sous forme de scripts, ou même interactive dans l'interpréteur Python. Ceci implique la création d'une interface utilisateur (ensemble de fonctions publiques) cohérente. On identifie les trois étapes principales d'une utilisation :

1. Définition des tests par l'utilisateur, notamment :
   (a) Paramètres à passer à ometah (pour indiquer le problème, l'algorithme, etc.).
   (b) Nombre de runs.
   (c) Chemin d'accès à l'exécutable.
2. Lancement effectif des tests (de la suite de runs), aucune intéraction avec l'utilisateur.
3. Traitement des données automatisé.
   (a) Calculs statistiques.
   (b) Générations des graphiques.

3.1.4.2 Limitation du temps de calcul

Contrairement aux "informaticiens", qui privilégient souvent l'efficacité d'un programme, l'exhaustivité de ses fonctionnalités, au profit du temps de calcul, ce dernier facteur étant généralement négligeable étant donné la tâche à accomplir, on devra pour le module de test limiter au possible la durée des calculs : dans le domaine des sciences expérimentales on se trouve souvent confronté à des lots très importants de données à traiter, et chaque milliseconde d'un algorithme est précieuse, dans la mesure ou on aura une très forte redondance de commandes. De plus je travaille sur une machine peu récente (processeur à 500Mhz), ce qui m'incitera à considérer le facteur temps comme essentiel. Comme suggeré plus haut, on pourra être confronté "en pratique" à une suite d'une dizaine de tests, effectuant chacun cinquante runs, chaque run pouvant réaliser jusqu'à cinq-cent mille évaluations, donc un nombre total d'évaluations de l'ordre de $1e+07$. Sachant qu'on devra traiter chaque point de l'échantillon à chaque itération de chaque run pour chaque test, on comprend vite l'interêt d'optimiser l'implémentation des optimisations (*sic*). On reviendra ultérieurement sur autre aspect plus en détail.

3.1.4.3 Compatibilité

Les fonctions propres à Linux devront être bannies, notamment les appels systèmes d'entrées / sorties, et les commandes de gestion des fichiers, on pourra pour celà utiliser les fonction du module os de Python, assurant leur universalité (relative).

3.1.5 Outils

Etant donnée le grand nombre de possibilités techniques offertes pour la conception d'un programme, il est essentiel de choisir les bons outils ; système d'exploitation, langage, etc., et la justification de ces choix trouve donc sa place dans un rapport.

Python Python est un langage relativement récent (en fait, récemment populaire), apprecié pour son utilisation sous forme de scripts, et pour le traitement de données, issues d'un flux quelconque, notamment d'un lourd programme en C / C++. Le post-traitement de données peut faire appel à un grand nombre de fonctionnalités, pour celà Python offre quantité de modules (entité extérieures utilisables à l'aide de la macro *import*), définissant chacun un espace de nommage pour minimiser la confusion dans l'utilisation. D'une grande souplesse dans sa syntaxe, Python a l'avantage (contrairement à Perl), de produire un code très lisible, grâce à sa politique de définition des blocs selon l'indentation des lignes. Il réduit de plus la quantité de code produite, les études montrent que pour un même programme, le code Python est inférieur de près de 30% à celui d'un programme C++. Semi-typé, non-déclaratif, orienté objet, très modulaire, simple de prise en main, compatible d'un OS à un autre : c'est le langage idéal pour réaliser le module de test où on emploiera des fonctions très variées, des entités claires seront manipulées, et les entrées / sorties fréquentes. En plus il est open source.

R / rpy Si nous nous limitions à la générations de quelques graphiques simples, un logiciel comme gnuplot aurait été suffisant pour "plotter" les données conservées dans un fichier. Mais on avait besoin d'un logiciel capable de réaliser toute sortes de graphiques, hautement paramètrable, et il fallait générer un ensemble de statistiques, dont l'implémentation aurait été fastidieuse. R est très complet pour le traitement statistiques de données, et permet de créer plusieurs types de graphiques, malheureusement au prix d'une syntaxe souvent obscure. Le gros avantage qu'offre R est la possibilité de son utilisation directe dans un script Python via la librairies de modules rpy, offrant l'interface entre les deux environnements. Un autre inconvénient de R est sa lourde occupation mémoire, d'où le délai particulièrement long à l'import des modules rpy.

Autres On mentionnera l'utilisation des librairies Numeric et LinearAlgebra, ayant facilité les opérations matricielles. Le module psyco est facultatif, mais recommandé : il permet d'optimiser les performances de l'interpréteur Python, les gains de temps étant très précieux pour notre programme. Toutefois on a constaté que les bénéfices de psyco n'étaient pas systématiques, en ce sens qu'il pouvait avoir l'effet inverse, soit ralentir le programme. A cet égard nous invitons l'utilisateur à effectuer des tests de performances, en affichant simplement le temps d'execution (ou en lançant les démonstration du fichier demo.py, affichant le temps de calcul final) avec, et sans psyco, pour en déduire l'utilité du module.

3.2 Récupération des données

3.2.1 Interêt du XML

Au lieu d'une sortie brute d'un flot de données chiffrées pour l'ensemble des points, on formatte la sortie en créant un type de document XML adapté à notre sortie, qui permettra la division de celle-ci en plusieurs parties, notamment une en-tête livrant les informations descriptives du test lancé. XML étant un méta-langage fortement modifiable, on l'adapte à notre problème en créant des balises explicites, et une structure cohérente et rigoureuse, en tâchant de ne pas trop approfondir l'arbre, ce qui complexifierait son "parsing".

3.2.2 Parsing avec le module xml

Le module xml de Python offre une interface relativement simple pour lire les données d'un arbre XML, avec les fonctions du sous-module *xml.dom.minidom*. Rappellons que DOM (*Document Object Model*) définit un standard pour la manipulation de données XML, avec un ensemble d'objets correspondant aux différentes entités XML (noeuds, attributs, etc.). On a donc en premier lieu utilisé ce module pour "parser" et récupérer les données de notre arbre XML, de structure relativement simple (peu d'éléments sont munis d'attributs, et l'organisation est fortement redondante), on devait tout de même explicitement définir le parcours à travers les noeuds de l'arbre sous forme d'une embrication de boucles, une pour chaque sous-élément en règle générale, les feuilles représentant les données, on affecte directement ses dernières dans les structures appropriées.

3.2.3 Réecriture du traitement

Conséquence de la compatibilité du module xml avec les standards courants, et de l'adaptation à tout type d'arbre, l'analyse rigoureuse du document et la récupération des données est assez longue relativement à notre problème. Une

fois l'ensemble du programme opérationnel, un traçage des opérations démontre que la partie la plus chronophage se trouve être le traitement du XML (soit la conversion de l'arbre XML en donnée dans nos structures Python). Etant donnée l'attention particulière à porter aux temps d'execution, et la répétition des tâches multipliant les gains de temps ponctuels (pour un test de N runs on aura N traitements du XML, un gain d'un dixième de seconde dans le traitement fera donc gagner vingt-cinq secondes pour une série de cinq tests de cinquante runs chacun), l'optimisation du traitement XML serait un gain précieux. Les gains apportés par un usage optimal des fonctions du module xml sont négligeables, et on se trouve limités par l'utilisation d'un module externe. La solution évidente est donc de réecrire un code de traitement de l'arbre XML, quitte à être moins rigoureuse, les bénéfices seront certainement importants.

Le nouveau code est crée dans un module *qparser* (*quick parser*, même si il ne "parse" pas réellement), remplaçant le parser précédent, et le fonctionnement du traitement est relativement naïf et simple, mais bien plus rapide : on lit séquentiellement la sortie XML, en identifiant les éléments à l'aide de la méthode *find()* de la classe *String* (bien plus rapide que le *string.find()* du module *string*, obsolète), recherchant les sous-chaînes d'une chaîne de caractères, en lisant le fichier ligne par ligne. L'inconvénient de cette méthode est qu'elle ne s'appuie pas sur l'organisation en arbre de la sortie, et se contente de lire "bêtement" la sortie selon la syntaxe utilisée. Elle est donc sensible à toute modification dans l'information de sortie, un saut de ligne inattendu par exemple. Mais ce sacrifice de l'élégance et la rigueur permet de doubler la vitesse d'execution d'un run, et ce gain de temps s'avèrera fort précieux pour les tests plus lourds, si le nombre d'évaluation avoisine la centaine de milliers en l'occurence.

3.3 Fonctionnement

3.3.1 Structure du calcul

Afin de pouvoir tester plusieurs algorithmes sur l'ensemble des problèmes implémentés, sans avoir a recréer une interface explicite pour chaque metaheuristiques, on utilise une interface commune, en utilisant notamment les méthodes virtuelles d'une classe C++ abstraite pour une métaheuristiques. Parmi les caractèristiques génériques ont retrouve la structure du calcul, organisé en itération d'une même arborescence, chaque entité disposant de caractèristiques communes, et on en déduira une structure dans le document XML. On peut ainsi décrire brièvement la structuration d'une optimisation, d'un point de vue simplement formel sans s'attacher à l'algorithmique associée, qu'on présentera éventuellement à part : durant une optimisation, on effectue un certain nombre d'itérations, chacune étant divisée en plusieurs pas de calcul (élément <*step*>), où plusieurs évaluations sont réalisées. Chaque pas contient un ensemble de points, solutions potentielles, dont on tente d'optimiser l'évaluation par la fonction du problème. L'itération stoppe quand un nombre donnée d'itérations à été

atteint, ou si la précision recquise est obtenue pour un des points de l'échantillon. L'entité définie dans le module ometahtest, la classe *Test*, est un ensemble d'optimisations (runs), au nombre de vingt-cinq par défaut, à partir de la même ligne de commande. On pourra alors étudier le comportement statistique à l'aide des fonctions du module ometahstats, en comparant éventuellement avec d'autres tests. Le nombre de runs est modifiable, avec une méthode de la classe *Test*, et, s'il ne conditionne pas directement la valeur de l'optimum obtenu (pas directement, car plus on lance de runs, plus on aura de diversité dans les résultats et par conséquent on aura plus de chances d'obtenir une bonne solution), permet une étude statistique plus fiable. Le contrepoids de ce gain est évidemment la durée de calcul accrue, linéairement en fonction du nombre de runs. La mémoire occupée s'en verra également accrue, on conserve tous les points dans une structure appropriée pour pouvoir les étudier par la suite.

3.3.2 Génération d'aléatoire

Souhaitant obtenir la plus grande variété de résultats possibles, on s'appuiera sur un générateur de nombres pseudo-aléatoires, génèrant une suite de nombres à partir d'une graine (*seed*). C'est donc de cette dernière que dépendra l'aléa, pour une série de plusieurs runs (chaque test comportant au moins une vingtaine de runs), et on devra minimiser la probabilité de redondance de la graine. On utilise donc un phénomène aléatoire pour la génération de la graine, à savoir le nombre de millisecondes écoulées depuis la dernière seconde. Mais ceci réduit à mille le nombre de graine possible, trop peu. Travaillant sous Linux, je modifie le générateur pour que la graine soit le produit de ce nombre de millisecondes avec le PID du processus d'ometah, c'est acceptable, mais ne fonctionnera pas sous Windows, et comme on souhaite une équivalence des performances, le produit par le PID seulement dans le cas d'une machine Unix (avec des conditions *#if!WIN32*) ne conviendra pas non plus. Comme on ne veut pas de graines ayant toutes de grandes valeurs (ie $> 1e10$), on choisit finalement la formule de calcul suivante : la graine sera égale à la somme de la différence de millisecondes entre l'instanciation de la métaheuristique et l'appel de la méthode initialisant l'aléatoire, et (seconde opérande de la somme) du produit du nombre de millisecondes écoulées depuis la dernière seconde par le modulo à cent du nombre de secondes comptées depuis le premier janvier 1970 ("since the epoch"). Ces modifications sont apportées dans le code C++ du fichier *itsMetaheuristic.cpp*.

Ce générateur de graine ne sera pas utilisé si on précise en ligne de commande la graine à utiliser (option *-R*, ou *–random-seed*).

3.3.3 Critères d'arrêt

Le calcul des métaheuristiques est soumis à deux critères d'arrêts. Le premier, naïf, est le nombre d'évaluations : on calcule jusqu'à avoir atteint ou dépassé le

nombre donné d'évaluations (une évaluation est un calcul de la fonction d'évaluation pour un point), l'optimum résultant de l'opération est le point bénéficiant de la plus petite valeur. On pourra dépasser le nombre d'évaluations requis si ce n'est pas un nombre multiple de la taille de l'échantillon, car le calcul est structuré en une suite d'itérations, chacune opérant sur un échantillon. Le second critère, très simple également, concerne la précision à atteindre, ainsi si un point s'approche suffisament de l'optimum (connu) en valeur, le calcul sera arrêté et on retiendra le point comme optimum. Pour chaque optimisation l'algorithme sera muni des deux critères d'arrêts, et non l'un ou l'autre. La mise en oeuvre du second critère a posé des problèmes dans l'implémentation, relatifs au fait que lors d'un test, les runs n'auront pas tous forcément effectué le même nombre d'évaluations, donc d'itérations, et comme plusieurs graphes portent sur l'évolution sur les itérations, on doit veiller à structurer correctement la sauvegarde des points pour ne pas subir de bugs conséquents à l'absence de valeurs pour un des runs, car il faut également considérer le cas où on trouve l'optimum "du premier coup". On a également du apporter des modifications pour ne pas tenir compte de la précision dans le cas d'un optimum inconnu, donc non stipulé dans la sortie XML, un bug se produisant car on admettait que chaque problème avait un optimum connu.

3.4 Représentation des points

3.4.1 Génération des graphiques

La génération des graphiques se fait simplement en appellant la méthode *plot()* du module *ometahstats*, en passant comme paramètres une liste des chemins d'accès vers les répertoires des tests à étudier. Chaque graphique fait l'objet d'une fonction du module *ometahstats*, on aura au préalable récupéré et structuré les données (ensembles de points), pour s'en servir lors des créations des graphiques.

Les sorties sont au format postscript, standard dans la communauté scientifique, et bien plus rapidement généré qu'une image au format PNG par exemple. Une conversion en PDF ou EPS est ensuite aisée (outils *ps2pdf*, *ps2epsi*, *eps2eps*, etc.).

3.4.2 Critères d'évaluation considérés

Afin d'étudier du mieux que possible des résultats, les graphiques se doivent d'afficher des informations pertinentes dans le plus grand nombre de dimensions possible, c'est à dire informer sur plusieurs critères ou mesures, par exemple donner les extrema d'un ensemble de points, ainsi que la variance des valeurs, soit refléter les critères de tests établis plus haut. Dans la mesure du possible on regroupera les données sur un même graphe, mais chaque test n'ayant pas

forcément le même nombre d'itération que les autres, ou le même ordre de grandeur sur une des dimensions représentées, il ne sera pas toujours judicieux de ne produire qu'une graphique, notamment pour les convergences. On tentera également de faciliter la lecture et l'interprétation en présentant des motifs non ambigües, en général dans un espace à deux dimensions.

3.4.3 Calcul du taux de réussite

On considèrera comme réussi un run ayant trouvé un point satisfaisant la précision requise relative à l'optimum connu, comme mentionné en 3.3.3, la précision constitue un critère d'arrêt, et les réussites seront donc des runs n'ayant pas atteint leur nombre maximal d'évaluations. La précision est un paramètre à donner en ligne de commande, avec l'option *-P* (ou *–precision*), et correspond à la marge d'erreur maximale tolérée par rapport à la valeur optimale. Celà signifie qu'un point pourra être considéré comme solution si son évaluation est suffisament proche de celle de l'optimum réel, sans qu'il en soit proche "géographiquement" (grande distance quadratique entre chaque ensemble des coordonnées). Par exemple si on a une cavité constituant un minimum local, mais euclidiennement éloignée dans l'espace vectoriel des solutions, si au plus profond de cette cavité l'évaluation du point est suffisament proche de celle du minimum global, cet extremum local sera accepté.

On parlera donc du taux de réussite pour un test, et ce taux sera égal au nombre de runs réussi divisé par le nombre total de runs effectués. Ce rapport est d'abord affiché sous forme de graphe, avec en abscisse l'index de chaque test, en ordonnée son taux de réussite en pourcentage. Les points étant au départ reliés, on préfèrera par la suite les présenter sous forme de "Diracs", pour souligner l'indépendance entre les tests, et donc entre leurs taux de succès.

Le fichier correspondant est *success_graph.ps*.

Exemple

Taux de succès pour les tests munis des paramètres suivants (50 runs chacun) :

1. *-e 50 -p Griewank -m CEDA -P 0.000001 -d 2*
2. *-e 100 -p Griewank -m CEDA -P 0.000001 -d 2*
3. *-e 200 -p Griewank -m CEDA -P 0.000001 -d 2*
4. *-e 500 -i 150 -p Griewank -m CEDA -P 0.000001 -d 2*
5. *-e 100 -i 1000 -p Griewank -m CEDA -P 0.000001 -d 2*

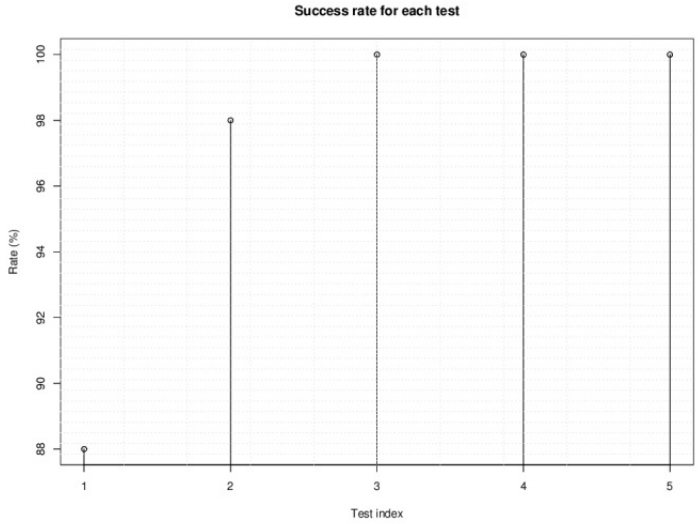


FIG. 3.1 – Affichage des taux de réussite pour une série de cinq tests.

3.4.4 Espace des solutions

3.4.4.1 Interêt

Dans la plupart des graphes, on s'intéresse seulement à la valeur des points. Il est aussi intéressant de présenter la localisation des points dans l'espace de recherche, et leur proximité relative de l'optimum. On pourra par exemple identifier les zones où se trouve un minimum local, et ainsi justifier une faible précision. On va donc cette fois comparer, non la valeur relative des points, mais leurs distance euclidiennes à l'optima, et entre eux. La représentation sera assez intuitive, car à partir de la représentation de la fonction on pourra localiser les points, par contre on perdra l'information statistique précise sur la distribution des valeurs, et l'évolution temporelle, car on représentera sur un graphe les N optima d'un test, où N est le nombre de runs effectués. Pour une meilleure précision, le plan ne rendra pas compte de tout l'espace de recherche, on affichera seulement l'espace délimités par les optima obtenus, et par l'optima réel, affiché comme réference. La représentation dans un plan est triviale pour un problème à deux dimension, chaque coordonnée correspond à un des deux axes du repère. Pour les cas d'une seule dimension on adopte une forme un peu particulière, à partir du vecteur trié des solutions, et pour plus de deux dimensions on représenteras toujours en deux dimensions, après avoir fait subir au points "une réduction" de dimensions, grâce à une analyse en composantes principales.

Une alternative envisagée était premièrement d'afficher dans un espace à trois dimensions les positions des points, leur troisième coordonnée (la "hauteur") représentant la valeur des points, on aurait abouti à une version morcellée des graphiques de problèmes présentés plus haut. Cependant R n'autorisait pas une telle représentation en pratique, il aurait fallu livrer les vecteurs ordonnés sur chaque coordonnés, ce qui nous était peu réalisable. Seconde alternative, l'affichage sour forme d'image, où chaque point du plan a une coloration virant vers le rouge proportionellement à l'intensité de sa valeur, permet de simuler une troisième dimension, et est une alternative à la vue en perspective. Cependant le problème est le même que précedemment, : la fonction R associée (*image()*), prend notamment en paramètres la liste des points en abscisses, et celle des points en ordonnées, imposant l'ordre des valeurs dans ces deux listes.

Le fichier correspondant à cette représentation est nommé *solutions_space.ps*.

Exemples Pour le problème Rosenbrock, on localise clairement la "vallée" où se trouvent des optima locaux, d'où la difficulté à approcher l'optimum réel :

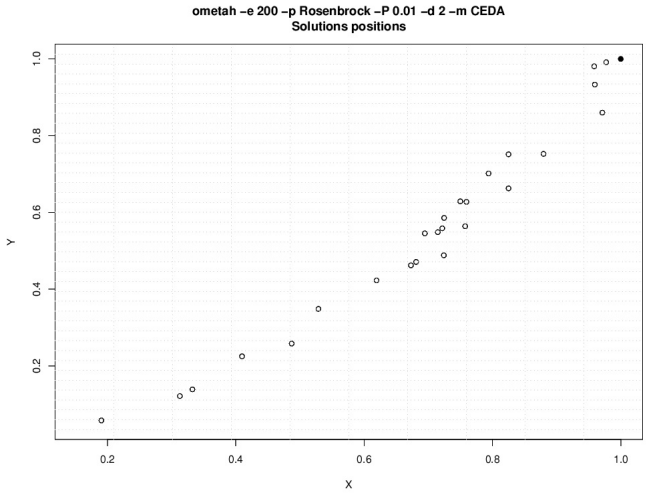


FIG. 3.2 – Espace des solutions pour l'algorithme CEDA sur le problème Rosenbrock.

Affichage pour un problème à une dimensions (figure 3.3) : le vecteur des optima est trié selon la valeur de chaque point, et la ligne représente la valeur réference, celle de l'optimum, l'écart vertical est donc la distance de chaque point à l'optimum sur l'espace droite. On n'aura pas forcément une courbe croissante, si des optima se trouvent des cavités locales, de meilleure valeur qu'un certain voisinage de l'optimum réel.

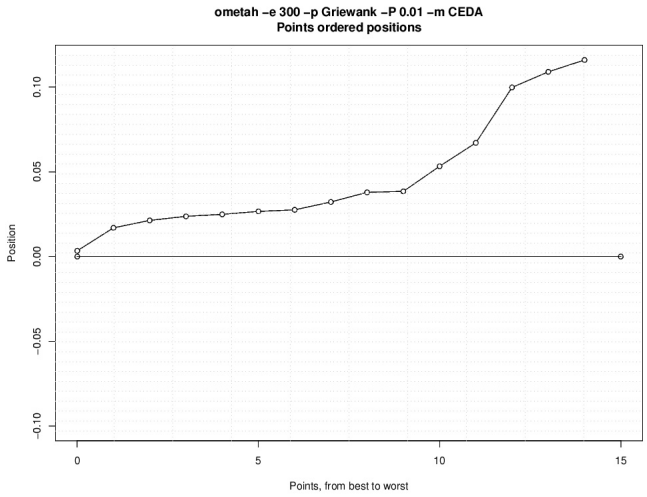


FIG. 3.3 – Affichage des solutions pour un problème à une dimension.

3.4.4.2 Analyse en composantes principales (ACP)

On s'est pour l'instant limités à étudier les valeurs des optima potentiels, les résultats de la fonction d'évaluation, en les représentant dans le plan sous forme de graphe, boîtes, ou histogramme, en modifiant la structure de l'ensemble de points en conséquence : les boîtes de convergences sont obtenues à partir d'une suite de liste, chacune contenant les points d'une itération différente, et ceci pour chaque test, on obtiendra un document contenant plusieurs graphiques si on considère un ensemble de plusieurs tests. Mais pour le graphes des valeurs, on s'est limités à ne conserver que le meilleur des optima sur l'ensemble des runs pour chaque test, d'où un graphe avec autant de points que de tests considérés. L'étude des solutions - les coordonnées des points - fait intervenir un nouveau problème. Les fonctions d'évaluation ne sont pas limitées à un espace en une dimension, et il est courant de considérer plusieurs dizaines de dimensions, les valeurs associés étant par exemple des paramètres du problème. On devra alors avoir recours à une "réduction de dimensions" si on souhaite représenter des points dans un espace plus réduit, notamment à trois dimension. Pour celà on peut utiliser la méthode statistique d'analyse en composantes principales (ACP), déduisant directement les valeurs dans l'espace de dimension réduite à partir de l'ensemble de points à n dimensions. Une autre solution envisagée est de considérer les valeurs propres (*eigenvalues*) de la matrice variance-covariance de nos points. On peut alors soit utiliser une des fonctions de R, si possible, ou bien effectuer l'algorithme "à la main".

La méthode d'analyse des composantes principales est implémentée dans R

(fonction *princomp()*), et la première idée, pour aller au plus simple pour une même efficacité, et ne pas réinventer la roue une énième fois, a évidemment été d'utiliser *princomp()*, via l'interface offerte par rpy, en invoquant *r.princomp()*. Seulement il s'est avéré impossible d'utiliser *princomp()* via rpy, que ça soit dans sa version émulée, ou bien en appellant directement la commande R avec *r(<chaîne de la commande>)*. L'alternative choisie a donc été d'implémenter l'ACP manuellement, ceci permettant de comprendre l'algorithme utilisé, consistant en un ensemble d'opération matricielles simples. On utilisera certaines fonctions offertes par les modules Numeric et LinearAlgebra. Pour aboutir aux résultats obtenus on peut procéder de plusieurs façons, on choisit celle proposée par la page Wikipedia consacrée (et vérifiée sur d'autres sources plus fiables), qu'on adapte à notre problème :

1. Organiser nos coordonnées dans une matrice, chaque ligne décrivant un point. Soit cette matrice D, de dimension $n \times dim$.
2. Calculer la moyenne sur chaque dimension, on a le vecteur M, de dimension $dim \times$ 1d x 1.
3. Centrer la matrice de départ, en soustrayant M à chaque ligne de D, soit cette matrice centrée S.
4. Calculer C, la matrice de covariance de S, symétrique, de dimension $dim \times dim$.
5. Calculer les vecteurs propres de C, et les ordonner selon les valeurs propres, de façon décroissante. Soit V la matrice obtenue.
6. Sélectionner les k premières colonnes de V, pour réduire les données à k dimensions, avec $1 \leq k \leq dim$.
7. A partir d'un point X (vecteur $1 \times dim$), on obtient le vecteur projeté Y $(1 \times k)$ en effectuant l'opération : $Y = P^t.S$.

Ainsi la méthode crée une nouvelle base de dimension réduite dans l'espace, en considérant les axes qui portent le plus de variance (poids des axes). Pour les problèmes considérés, on applique la même fonction sur chaque dimension, et on n'aura généralement pas de forte variance sur les dimensions, les poids des axes ne varieront pas beaucoup, et de ce fait la réprésentation en dimensions réduite (M) d'un problème à N dimensions tendra à être identique à celle de ce même problème en dimensions M.

On appliquera cette méthode pour réduire les points de dimensions supérieures à deux (jusqu'à 50), à deux dimensions seulement pour afficher les points dans l'espace des solutions.

3.4.5 Comparaison de distribution

On souhaite comparer des distributions de valeurs, notamment les erreurs de chaque point relatives à l'optimum, ceci pour chaque test. Graphiquement on

peut construire un histogramme des fréquences pour chaque test, décrivant le nombre d'entités dans chaque intervalle. Si on se trouve avec plusieurs de ces histogrammes, et qu'on souhaite comparer rigoureusement les distributions associées, on utilisera de préférences des tests statistiques, en particulier ceux basés sur l'analyse de variance (ANOVA) pour la comparaison de plusieurs échantillons. Ayant au minimum deux échantillons à comparer, on utilisera selon la situation le test de Mann-Whitney (deux échantillons), ou celui de Kruskal-Wallis (au moins trois échantillons). Ces tests sont dits non-paramétriques, car ils ne considèrent pas les paramètres de la distribution (valeurs des points), mais se basent sur un test du rang des valeurs des échantillons, en faisant l'hypothèse de l'identité des distributions.

Ces tests sont implémentés dans R (*wilcox.test* pour le M-W, *kruskal.test* et pour K-W). Mais on rencontre le même problème que pour l'ACP, à ceci près que dans ce cas ce n'est pas un obscur bug qui apparaît, mais seulement l'information que cette fonction n'est pas reconnue. Donc une fois de plus, après le parser XML et l'ACP, on devra certainement reprogrammer ces tests.

Je n'ai finalement pas eu à réimplémenter les tests, il suffisait seulement de savoir que le séparateur entre module R (un point) devenait "_" sous rpy, il est donc possible d'invoquer *r.wilcox_test* et *r.kruskal_test* à partir de rpy. La valeur renvoyée est un dictionnaire (une structure Python, équivalente à une table de hachage Perl), et on s'interesse principalement à la valeur déterminant l'identité des distributions (*p-value*). Au-delà d'une limite donnée (généralement 0.05), on peut estimer que les distributions sont équivalentes, autrement elles diffèrent. Cette valeur est peu significative en elle-même, il n'y a de toute façon pas lieu de produire un affichage graphique car on procède au test non-paramètrique pour l'ensemble des tests réalisés, on n'a donc qu'une seule valeur, qui aura alors sa place dans le rapport généré. On affiche la valeur du test non-paramètrique sur l'ensemble des distributions (on utilisera donc Kruskal-Wallis dès trois tests), et les tests de Mann-Whitney sur les tests deux à deux.

Exemple Des distributions des optima pour des tests de vingt-cinq runs, selon l'amplitude des valeurs, on n'aura pas forcément le même nombre de subdivisions (*breaks*) :

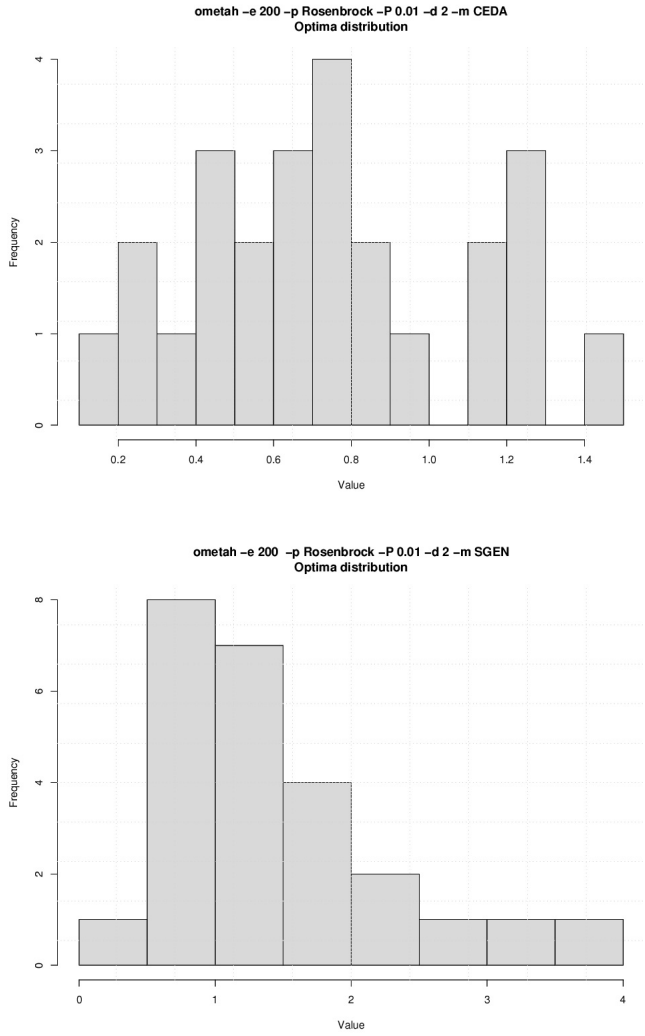


FIG. 3.4 – Distribution des optima.

3.4.6 Convergence des valeurs

Pour étudier la progression des valeurs obtenues, on affiche une représentation de la convergence des valeurs ; en effet les algorithmes garantissent parfois une convergence mathématique vers l'optimum, en ce sens qu'on convergera forcément vers l'optimum global en temps infini. Par exemple l'exploration aléatoire de l'espace de recherche converge, car la probabilité de trouver l'optimum en temps infini temps vers l'unité. On considèrera également la notion de conver-

gence de l'apprentissage, au sens où les valeurs tendent à diminuer progressivement, pour se fixer à une valeur donnée (optimum local ou global.. Le graphique qu'on associe à cette notion de convergence dans le temps discrètise ce dernier en considérant l'itération comme unité de temps, et au lieu de simplement afficher un graphe avec un ensemble de points reliés entre eux, on utilise la sémantique des boîtes "à moustaches", ou boîtes quantiles (*quantile boxes)*, et on considèrera deux types de graphes :

1. Pour les optima, en comptabilisant seulement l'optimum de chacun des runs, pour chaque itération.
2. Pour tous les points, de chaque run, pour chaque itération (on atteindra vite un volume considérable de points, ce qui expliquera un certain ralentissement du programme).

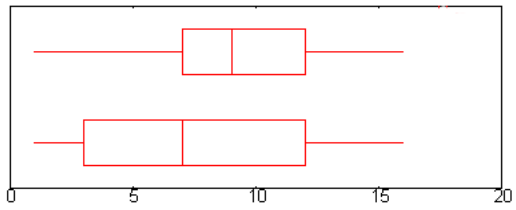


FIG. 3.5 – Modèle de boîte à moustache.

Chaque *quantile box* correspondra à un échantillon, ce dernier étant pour la première situation l'ensemble des optima (un pour chaque run) à une itération donnée. De même pour le second cas, chaque boîte modélisera l'échantillon de l'ensemble des points (tous ceux de l'échantillon de chaque run) à une itération donnée. L'utilisation de ces boîtes n'étant pas forcément familière au lecteur, on citera la définition donnée par l'article de Wikipedia :

"Ce diagramme résume seulement quelques caractéristiques de position du caractère étudié (médiane, quartiles, minimum, maximum ou déciles). Il est utilisé pricipalement pour comparer un même caractère dans deux populations de tailles différentes. Il s'agit de tracer un rectangle allant du premier quartile au troisième quartile et coupé par la médiane. Ce rectangle suffit pour le diagramme en boîte. On ajoute alors des segments aux extrémités menant jusqu'aux valeurs extrêmes, ou jusqu'aux premier et neuvième déciles (D1 / D9), voire aux 5e et 95e centiles. On parle alors de diagramme en boîte à moustaches ou de diagramme à pattes."

La taille des moustaches (*whiskers*) étant limitée, R affiche des points si des valeurs de l'échantillon sont trop excentrées : *"Dans les diagrammes en boîte de Tukey, la longueur des « moustaches » vaut 1,5 fois l ?écart interquartile."*

Graphiquement, la convergence de l'algorithme correspondra à une régression globale des positions des boîtes, et on pourra parfois constater une brusque

augmentation, pour ensuite diminuer à nouveau, quand la recherche se concentre dans une autre zone de l'espace de recherche, par exemple lors de l'extraction d'un zone de minimum local, vers une autre cavité.

Comme défini dans [6], on affiche le graphe de la convergence de l'erreur médiane, et non les boîtes des échantillons (si la valeur de l'optimum est nulle, l'erreur est équivalente à la valeur des points).

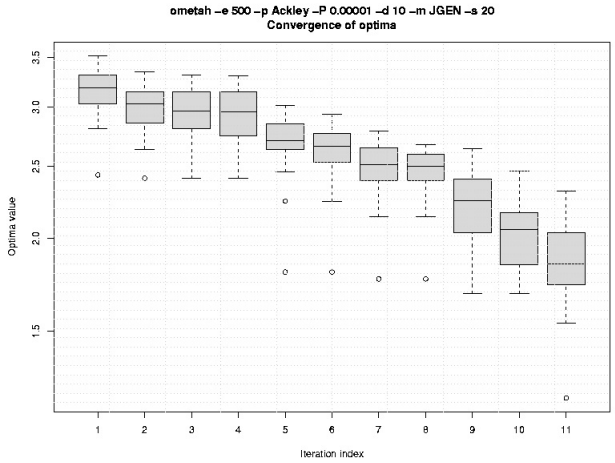


FIG. 3.6 – Convergence des optima pour un test de 30 runs.

3.4.7 Graphe des valeurs

Peut-être le graphe le plus aisément compréhensible, car il est naturel d'afficher les valeurs obtenues, on aura donc l'indice de chaque test en abscisse, et on affichera trois valeurs : le meilleur optimum obtenu (autrement dit le meilleur run), le pire (celui du moins bon run), et la valeur médiane sur l'ensemble des runs. Cette dernière sera sans doute la valeur la plus pertinente à considérer, car le pire et le meilleur sont des phénomènes ponctuels, et n'ont pas vraiment de valeur statistique car il suffit d'un "coup de chance" (aléatoire oblige) pour qu'un algorithme moyen obtienne une valeur plus que correcte, alors que sa valeur médiane sera sensiblement moins bonne que pour d'autres algorithmes. On trouvera ce graphe sous le nom *graph_ optima.ps*. On donne un exemple d'affichage sur la figure 3.7, où on peut constater les phénomène évoqué ci-dessus : chaque test, de 1 à 4, correspond à un algorithme différent pour le même ensemble des autres paramètres, et on voit que le premier algorithme obtient un très bon meilleur optimum, mais on peut supposer que celà ne soit pas du à l'efficacité de l'algorithme, mais seulement à un bon tirage (c'est l'algorithme évolutionnaire simple), car la valeur médiane est beaucoup moins bonne que celle de l'algorithme suivant (évolutionnaire expérimental).

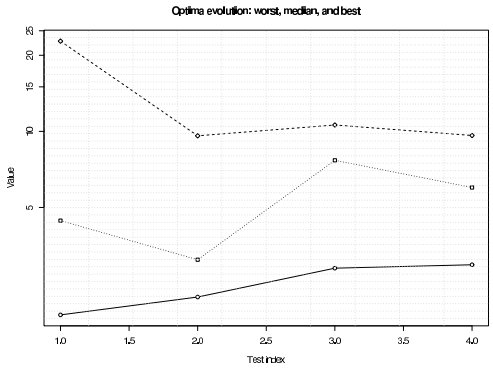


FIG. 3.7 – Graphe des valeurs.

3.4.8 Rapport

En sortie du programme, en complément des graphiques, on génère un rapport textuel contenant les caractèristiques essentielles des tests lancés (problème, métaheuristiques, et paramètres), et les résultats statistiques (moyenne et écart-type) et optima obtenus. TEX étant un standard dans la communauté scientifique, on le préfèrera à une sortie en texte brut peu élégante, et inappropriée à une conversion simple et rapide en d'autres formats (postscript, PDF, HTML, ou DVI) comme le permet TEX.

OMETAHLAB REPORT

ometah -e 200 -p Rosenbrock -P 0.01 -d 2 -m CEDA
ometah -e 200 -p Rosenbrock -P 0.01 -d 2 -m NMS
ometah -e 200 -p Rosenbrock -P 0.01 -d 2 -m SGEN
ometah -e 200 -p Rosenbrock -P 0.01 -d 2 -m JGEN

Problem	Algo	Evaluations	Precision	Optimum	Mean	Std	Success Rate
Rosenbrock	CEDA	200	0.01	0.164	0.758	0.360	0
Rosenbrock	NMS	200	0.01	0.015	1.345	1.898	0
Rosenbrock	SGEN	200	0.01	0.072	1.291	0.602	0
Rosenbrock	JGEN	200	0.01	0.184	1.232	0.510	0

Non-parametric test over optima errors 0.001944

FIG. 3.8 – Exemple de rendu PDF du rapport.

3.5 Optimisation du programme

3.5.1 Optimisation du temps de calcul

Le programme ayant atteint une certaine complexité, une attention suffisante n'a pas forcément été portée durant l'implémentation aux détails faisant que l'utilisation de telle fonction s'avèrera plus efficace qu'une autre, plus généralement à l'optimisation de la vitesse d'execution. Et étant donnée la nature du programme, il serait également un comble de ne pas s'attarder un minimum sur l'optimisation du logiciel utilisé pour optimiser du mieux que possible. Il existe un ensemble de petits astuces, plus ou moins évidentes, glanées sur Internet, concernant les manipulations de chaînes de caractères et de listes pour optimiser le temps d'execution. Mais la première étape d'une optimisation est le profilage du code, dont l'objectif est l'identification des portions de codes les plus gourmandes, à optimiser en priorité. Il est clair que dans notre programme, dans l'état où cette optimisatin a été réalisée, la plus forte redondance est le traitement répétitif des sorties XML (dont on a déjà évoqué l'optimisation). Une fois les fonctions et segments de code identifiés, on peut commencer à effectuer quelques modifications : on minimiser l'usage inutile de variables intermédiaires, les imports sont positionnés stratégiquement (dans une fonction ou en entête du module), minimiser le nombre d'accès disques, etc. et d'autres propres à Python, comme l'utilisation du module *cPickle* au lieu de pickle, pour la serialization, qui effectue les mêmes tâches mais presque dix fois plus vite.

Toutefois, quel que soit le langage utilisé, la première et meilleure des optimisations à réaliser concerne les algorithmes employés, en s'interessant à leur complexité en temps et en espace, pour minimiser la durée du calcul et le remplissage de la mémoire. Ces deux aspects ne vont pas forcément de paire, et sont même souvent opposé : un gain sur le temps pourra être réalisé au prix d'un plus grand espace mémoire occupé. Cinq heures de travail auront permis de gagner au total une seconde sur un même ensemble de tests, soit un gain de vingt-cinq pourcent, le rapport des durées n'est peut-être pas très convaincant, mais l'impact sur les tests de plus grande ampleur sera très impressionant.

3.5.2 Portabilité

Bien qu'étant développé sous un système Unix (Linux), sous une licence libre (LGPL), où la grande majorité des projets concerne essentiellement des plateformes Unix, nous devons assurez la compatibilité du programme avec Windows. Certains projets proposent plusieurs versions différentes pour Unix (et les différentes distributions Linux, BSD, etc.) et Windows, les sources faisant l'objet d'adaptation aux OS respectifs. Cependant l'envergure de notre projet étant relativement réduite, et la compatibilité se trouvant réalisable, en se limitant aux fonctions et libraires présentes dans les deux OS, on pourra se passer de la réalisation d'une version alternative spécialement pour Windows. Réalisant

que j'utilisais certaines librairies propres à Windows dans le code C++, et des appels systèmes Unix en Python dans ometahlab, quelques "purges" ont été nécessaires : dans ometahlab, les appels à *os.system()* pour executer une commande du shell, peu élégants, ont été remplacés par d'autres fonctions du module os, garantissant la compatibilité avec tout système. Dans ometah, le code C++ incluait les librairies *sys/types.h* et *unistd.h* pour l'appel à *getpid()* lors de la génération d'une graine pour le générateur pseudo-aléatoire. Des tests avec la balise *#if!WIN32* permettent d'utiliser les fonctions propres à Unix quand ce système et utilisé, sinon on n'inclue ni n'appelle les objets concernés.

3.5.3 Lisibilité

Le programme réalisé lors de ces deux mois sera soumis à réutilisation, adaptation et corrections ("Each new user of a new system uncovers a new class of bugs", Kernighan), et il est de toute façon impératif pour un programme de moyenne ampleur de réaliser un code lisible, pour minimiser la perte de temps lors de relectures. De plus le logiciel étant labellisé "open source", sous licence LGPL, on ne doit pas décourager nos futurs utilisateurs qui auront eu la bonté de s'interesser à notre travail. Rappellons tout d'abord quelques évidences, valable pour tout programme et la plupart des langages couramment utilisés. La réalisation d'un programme, implémentation d'algorithme, ou fonction quelconque, émane généralement d'une idée préconçue de la suite des opérations à effectuer. Selon la complexité du problème on peut être amener à repenser le programme - au sens d'une suite d'instructions - lors de son implémentation, l'interface du code supprimant un niveau d'abstraction, pour le modifier selon divers critères : adaptation à la syntaxe du langage (n'a pas lieu d'être en général si on maîtrise le langage), optimisation de la complexité de l'algorithme (ayant pour conséquence un moindre temps de calcul), optimisation matérielle (limitation en mémoire). Généralement le programmeur inexpérimenté se concentrera alors sur les performances du programme, car ayant désormais bien assimilé le fonctionnement de l'algorithme, l'attention portée au futur confort d'une relecture et recompréhension sera releguée au rang des travaux secondaires. Commentaires dans le code et indentation sont évidemment indispensables, les premiers devant être clair et concis, évitant la litteralisation des instruction (ie "on affecte la valeur 2 à x" pour l'instruction $x = 2$), le verbiage inutile, et leur systématisation, ôtant leur rôle annexe de détection des points délicats du programme. Contrairement à l'acceptation naïve de cet aspect de l'optimisation du code, celà ne constitue pas seulement la composante formelle d'une vision duale fond / forme, mais englobe l'ensemble des deux aspects, où un même algorithme peut être implémenté de façons plus ou moins compréhensives.

Python peut être particulièrement traître pour la réalisation d'un code lisible : sa syntaxe minimaliste, et sa structure par indentation garantissent un minimum de lisibilité pour la localisation des structures de contrôles et des blocs (même si on peut regretter l'absence des accolades à la C), mais le typage non-explicite et l'absence de déclaration des variables font courir le risque d'une confusion

dans les labels utilisés, et on doit être particulièrement attentif au nommage des variables, ou il est recommandé si possible de laisser deviner le type affecté à cette variable. Une vision alternative ironique est au contraire de maximiser la confusion du code, comme le conseille Roady Green dans le document How To Write Unmaintainable Code, sous-titré justement "Ensure a job for life;-)". Je doute cependant de la validité à long terme de cette stratégie, ou l'auteur même risque fort de se perdre dans l'ésoterisme de son code quelques jours après sa production, voir *http ://mindprod.com/jgloss/unmain.html*

3.5.4 Méta-optimisation

On pourra s'intéresser à optimiser une optimisation (*sic*), en recherchant un ensemble de paramètres minimisant le temps de calcul, et maximisant le gain (c'est à dire minimisant l'évaluation). Ceci peut s'apparenter à une optimisation multi-objectif, où les objets de l'optimisation sont contradictoires : on pourra obtenir une très bonne solution, au prix d'un long calcul, et inversement. Ceci rejoint le problème plus pratique de la précision, où on doit alors équilibrer ces deux critères en fonction de ses besoins : autant pour certaines applications une grande précision sera nécessaire, autant d'autres se satisferont d'une approximation moyenne, mais seront très exigeants sur le temps de calcul.

3.6 Utilisation

3.6.1 Configuration requise

L'utilisateur pourra se procurer Open Metaheuristic sur le site officiel (*http ://ometah.berlios.de*), soit à partir des sources, soit en téléchargeant directement les fichiers binaires. Dans le second cas seulement il n'aura pas de compilation à réaliser, donc scons et gcc (g++) ne seront pas nécessaires. Pour l'utilisation d'ometahlab, Python et R évidemment sont requis, ainsi que les modules rpy, Numeric, LinearAlgebra, un visualisateur de fichiers postscript peut aussi être utile pour afficher les graphes. Le module psyco est recommandé pour accélerer les calculs, mais non obligatoire. Pour le programme *ometah_registration*, réalisant l'optimisation d'un problème plus concret, le recalage d'image rétinienne, on aura besoin de la librairie CImg (*CImg.h*), aisément trouvable sur le réseau (*http ://cimg.sf.net*).

Au niveau du matériel, un fonctionnement basique ne requiert pas une configuration particulière, cependant les calculs seront d'autant moins longs qu'on dispose d'un processeur puissant, l'idéal pour un ordinateur personnel serait de disposer d'un processeur à 3Ghz gérant l'hyperthreading, et si on dispose de plusieurs machines, ou de clusters ordinateurs, il devrait être possible de parallèliser les calculs, mais nous ne fournissons aucune documentation à ce sujet.

3.6.2 Installation

On ne gère pas de procédure d'installation explicite, l'utilisateur n'aura qu'à aller dans le répertoire de ometah pour compiler avec scons, les modules Python ne nécessitent pas de compilation, ils sont à utiliser directement. Une compilation typique se fait simplement de la façon suivante :

$ ls
ometah ometahlab
$ cd ometah
$ scons
scons : Reading SConscript files ... Checking for C++ header file CImg.h... yes scons : done reading SConscript files. scons : Building targets ... g++ -Wall -ansi -pedantic -Wimplicit -Wredundant-decls -Wreturn-type -Wunused -ggdb -Wno-deprecated -c -o interface/omet ah.o interface/ometah.cpp g++ -Wall -ansi -pedantic -Wimplicit -Wredundant-decls -Wreturn-type -Wunused -ggdb -Wno-deprecated -c -o common/itsPoint.o common/itsPoint.cpp

[... patience ...]

g++ -o ometah interface/ometah.o -Lcommon -Lcommunication -Lmetaheuristic -Lproblem -Linterface -Lproblem/CEC05 -Lproblem/registration -Lmetaheuristic/estimation -Lmetaheuristic/random -Lmetaheuristic/sampling -Lmetaheuristic/neldermead -Lmetaheuristic/simplegenetic -Lmetaheuristic/jpgenetic -lcommon -lcommunication -lmetaheuristic -lproblem -lproblem_ CEC05 -linterface -lestimation -lrandom -lsampling -lneldermead -lsimplegenetic -ljpgenetic
scons : done building targets.

Une fois la compilation terminée il suffit de lancer ometah en ligne de commande, avec les paramètres désirés :

$ ./ometah
< ? xml-version="1.0" encoding="iso-8859-15" ?>
<ometah>
<problem>
<key>Rosenbrock</key>
<name>Rosenbrock</name>
<description>Rosenbrock is a classical test problem</description>
<formula>$R_ {n}(\overrightarrow{x})=\sum_ {i=1}^{n-1}\left(100\cdot\left(x_ {i}^{2}-x_ {i+1}\right)^{2}+\left(x_ {i}-1\right)^{2}\right)$</formula>
<dimension>1</dimension>
etc.

3.6.3 Création d'un script

Bien que toute personne familière d'un langage informatique soit en mesure de comprendre le principe en lisant quelques lignes du fichiers de démonstrations *demo.py*, ou bien l'exemple dans le fichier README, nous présenterons tout

de même, dans un souci de pédagogie, la création d'un script type, avec les explications nécessaires (les commentaires, précédés d'un dièse). Ci-dessous donc la description de la rédaction d'un script, et non la suite d'instruction dans l'interpréteur (syntaxiquement analogue, mais les détails dans la présentation diffèrent, ici on n'aura pas directement les sorties).

 # on importe les deux modules nécessaires, la convention est d'accorder une ligne à chaque module, mais on peut très bien écrire "import ometahtest, ometahstats"

import ometahtest
import ometahstats

 # on crée un nouveau test, nommé t, comme instance de la classe Test, présente dans ometahtest

t = ometahtest.Test()

 # on donne la chaîne de paramètre de ce tests, qui sont exactement les mêmes qu'on passerait directement à l'exécutable, avec la méthode setArgs (pour "set arguments"), ici on demande au test d'avoir un maximum de 230 évaluations, sur le problème Rastrigin, avec la méthode de recherche de Nelder-Mead, jusqu'à une précision de 0.01

t.setArgs('-e 230 -p Rastrigin -m NMS -P 0.01')

 # on ajuste le nombre de runs, 25 par défaut

t.setNbRuns(40)

 # on indique le chemin d'accès au binaire ometah, la chaîne donnée en exemple est la valeur par défaut

t.setOmetahPath('../ometah/ometah')

 # enfin, on démarre le test, à la fin l'instance de Test sera serializée dans un répertoire donné, pour réutilisation

t.start()

 # une fois le test réalisé, on peut réaliser la génération des graphes à partir de ses données, la fonction stat() prend en paramètre une LISTE de chemins d'accès, on peut ainsi réutiliser des tests déjà générés, la méthode getPath() de Test renvoie le chemin d'accès, les crochets signifient la qualité de liste de l'argument.

ometahstats.stat([t.getPath())

Pour être encore plus clairs, on présente un exemple, avec le compte-rendu des modifications apportées sur le système de fichiers (soit les répertoires et fichiers crées). Un petit script, *clean.py*, permet de nettoyer l'arborescence des résultat, autrement dit en supprimant tout le contenu du répertoire /results, la taille de celui-ci s'accroissant rapidement lors de tests conséquents. On l'aura auparavant

lancé (*python clean.py*), le répertoire *results/* est donc vide désormais (si il est supprimé par mégarde, ometahlab se charge de le recréer automatiquement).

Soit le script suivant, dans le fichier *script.py* :

```
t = ometahtest.Test()
t.setArgs('-e 200 -p Rosenbrock -P 0.01 -d 2 -m CEDA')
t.setNbRuns(25)
t.setOmetahPath('../ometah/ometah')
t.start()

u = ometahtest.Test()
u.setArgs('-e 200 -p Rosenbrock -P 0.01 -d 2 -m NMS')
u.setNbRuns(25)
u.setOmetahPath('../ometah/ometah')
u.start()

v = ometahtest.Test()
v.setArgs('-e 200 -p Rosenbrock -P 0.01 -d 2 -m SGEN')
v.setNbRuns(25)
v.setOmetahPath('../ometah/ometah')
v.start()

w = ometahtest.Test()
w.setArgs('-e 200 -p Rosenbrock -P 0.01 -d 2 -m JGEN')
w.setNbRuns(25) w.setOmetahPath('../ometah/ometah')
w.start()

paths = [ t.getPath(), u.getPath(), v.getPath(), w.getPath() ]
ometahstats.stat(paths)
```

On le lance dans la console, produisant la sortie standard suivante (les traits verticaux sont les barres de défilement remplies) :

```
$ python script.py

Running ometah -e 200 -p Rosenbrock -P 0.01 -d 2 -m CEDA
||||||||||||||||||||||||

Running ometah -e 200 -p Rosenbrock -P 0.01 -d 2 -m NMS
||||||||||||||||||||||||

Running ometah -e 200 -p Rosenbrock -P 0.01 -d 2 -m SGEN
||||||||||||||||||||||||

Running ometah -e 200 -p Rosenbrock -P 0.01 -d 2 -m JGEN
||||||||||||||||||||||||

Creating graphics and report... Results in results/results_1
```

Qu'aura produit ce lancement ? On trouve maintenant dans le répertoire *results/* (auparavant vide), les répertoires suivants :

- *results_1/* : les résultats de l'ensemble de tests, répertoire produit par *ometahstats.stat()*.
- *Rosenbrock_CEDA_d2_e200_r12182_1/* : les données relatives au test sur le problème Rosenbrock, avec l'algorithme CEDA, en dimension 2, etc. de même pour les répertoires suivants.
- *Rosenbrock_JGEN_d2_e200_r34894_1/*
- *Rosenbrock_NMS_d2_e200_r28904_1/*
- *Rosenbrock_SGEN_d2_e200_r14450_1/*

Le chiffre 1 à la fin du répertoire de chaque test sert à identifier les tests dans le cas où on aurait deux fois le même test lancés (donc deux fois la même graine aléatoire, ici 12182 pour le premier test). Examinons le contenu d'un répertoire de test :

$ ls Rosenbrock_CEDA_d2_e200_r12182_1/
run.log TEST

Le fichier *run.log* est un journal contenant l'optimum obtenu à chaque run du test, et TEST est l'instance de Test serializée, pour réutilisation par ometahstats.

Maintenant le répertoire de résultats globaux :

[jp@results]$ ls results_1/
convergence_error_all.ps convergence_optima.ps distribution_errors.ps graph_optima.ps solutions_space.ps testboxes_optima.ps convergence_error_opt.ps convergence_points.ps distribution_optima.ps report.tex success_graph.ps

Les fichiers d'extension *ps* sont les graphiques, au format postscript, *report.tex* est le rapport TEX, et on pourra génerer une version PDF par exemple avec la commande pdflatex.

Chapitre 4

Algorithmes

4.1 Estimation de distribution

Cette métaheuristique était déjà implémentée, j'ai juste étudié le principe de son fonctionnement, que nous décrirons brièvement : la version implémentée est Continuous Estimation of Distribution (CEDA), une des variantes appartenant à la famille des algorithmes à estimation de distribution (EDA), à l'origine une variante des algorithmes évolutionnaires. Le principe général est de tirer un ensemble de points au hasard, selon une loi de probabilité déduite de la distribution apparente des solution de l'échantillon courant, de façon à orienter les nouveaux tirages vers la "zone" a priori la plus intéressante. La principale difficulté est d'estimer la distribution à partir d'un ensemble de point, on en déduit ensuite aisément un tirage aléatoire conséquent. Notons que pour les problèmes continus, la distribution suit généralement une loi normale. On trouvera une étude détaillée des algorithmes à estimation de distribution dans [5].

On utilisera l'option *-m CEDA* en ligne de commande.

4.2 Algorithme aléatoire

La plupart des métaheuristiques se basant sur des algorithmes stochastiques, il seront quoiqu'il arrive plus efficaces qu'un algorithme totalement aléatoire. On implémente un algorithme aléatoire totalement stupide, qui se contente de choisir des points dans l'espace des solutions, selon une loi de probabilité uniforme. On pourra l'utiliser pour comparer avec des métaheuristiques plus intelligentes, pour être sûr que celles-ci sont plus efficace que l'aléatoire total. Evidemment l'algorithme aléatoire ne peut garantir aucune précision en un temps fini, mais convergera mathématiquement, au sens où on obtiendra une valeur atteignant la précision requise avec un nombre suffisant d'itérations. Attention, cet algorithme

renouvelle l'échantillon à chaque itération, et ignore les itérations précédentes. Un algorithme tirant à chaque pas un même nombre de points au hasard, en ne sélectionnant que les meilleurs de l'ensemble formé par ce nouvel échantillon et le précédent, s'apparenterait à un algorithme évolutionnaire simpliste, comme nous le verrons par la suite.

L'option à utiliser sera *-m RA* (*Random Algorithm*).

4.3 Echantillonage régulier récursif

Cet algorithme n'est pas réellement une métaheuristique, plutôt une méthode de recherche déterministe. Le principe est de réalise un échantillonage régulier de l'espace de recherche, en évaluant un nombre donné de points par dimension, de façon à ne sélectionner que les noeuds d'une grille (pour deux dimensions), ou d'un hypercube (pour plus de 3 dimensions). La récursivité de l'algorithme revient à faire un emboîtement d'autant de boucles que de dimensions, chacune itérant un nombre de fois égal à la résolution donnée sur cette dimension, autrement dit au nombre de points échantillonnés sur cette dimension. On n'aura pas le cycle apprentissage / diversification / intensification, comme c'est le cas pour un algorithme évolutionnaire ou une estimation de distribution, mais seulement un ensemble de solutions, dans la partie diversification, dont le cardinal sera égal à la quantité d'évaluations désirée, de laquelle on aura déduit la résolution de l'hypercube. Une contrainte s'impose : la taille de l'échantillon devra être multiple du nombre de dimensions du problème si on veut toujours avoir la taille voulue de l'échantillon. Autre conséquence : la taille de l'échantillon devra être au moins égale au nombre dimensions (la taille par défaut, dix, ne conviendra pas si on utilise trente dimensions pour notre problème).

Cette méthode n'a pas pour objectif de livrer le ou les optimum (optima) avec une grande précision, les coordonnées sur chaque dimension étant limitée à un multiple de la taille de l'intervalle, mais à donner une idée de la zone où peux se trouver l'optimum, ou bien à avoir une solution acceptable, sans être excellente. Cette méthode de recherche évacue les problèmes liés aux cavités d'optima locaux, mais est très limitée, même si on est sûr de trouver l'optima en faisant tendre la résolution vers l'infini, ceci revient à faire une recherche exhaustive, donc irréalisable en temps et espace raisonnables.

Une fois implémenté, un problème survient au lancement : un seul point se trouve dans l'échantillon, de coordonnée zéro en chaque dimension de l'espace, car en cette position, l'origine, se trouve pour certains problèmes l'optimum désiré. Cette position remarquable est due au centrage des problèmes (voir 2.3), et l'atteinte de l'optimum étant un critère d'arrêt, le calcul est interrompu dès le calcul du premier point. La solution est d'ajouter en paramètre dans la ligne de commande l'option '–precision -1', ainsi tous les points seront calculés.

On donnera donc les paramètres *-m GS -P -1* pour utiliser cette méthode (*Grid Sampling*).

4.4 Recherche de Nelder-Mead

C'est une méthode géométrique dont le principe est relativement simple : on construit un N-simplexe à partir de l'échantillon donné (N étant le nombre de dimensions du problème), et ce simplexe est soumis à certaines transformations conditionnées par l'ordonnancement des points selon leurs valeurs. La recherche simple modifie à chaque itération un seul point de l'échantillon, résultant d'une transformation dépendant des coordonnées du centroïde du simplexe (réflexion, expansion, contraction externe et interne), mais ne sera pas très intéressante pour notre recherche, où on désire diversifier l'échantillon. Pour celà on utilisera la version multidirectionnelle, telle que décrite dans [1], qui à chaque itération crée un nouveau N-simplexe, congruent au précédent. Le principe est le suivant :

Soit une famille libre de x_i, $1 \leq i \leq N+1$, de points de l'espace des solutions à dimension N, on peut donc voir ces $x_i$comme des vecteurs à de taille N, telle que :

$i \leq j \Rightarrow f(x_i) \leq f(x_j)$, ces $N+1$ points formant un N- simplexe, f étant la fonction d'évaluation du problème.

Définissons les trois opérations de base, applications de l'espace des solutions dans lui-même :

Réflexion $x_i^R = x_1 - (x_i - x_1)$
Expansion $x_i^E = x_1 - 2(x_i - x_1)$
Contraction $x_i^C = x_1 - 0.5(x_i - x_1)$

Avec $i = 1...N+1$

On pourra identifier ces opérations par leur coefficient μ dans l'équation :

$x_i^* = x_1 - \mu(x_i - x_1)$

On notera Sle simplexe formé par l'échantillon courant de points, S^* le nouveau simplexe calculé, et respectivement S^R, S^E et S^Cles simplexes résultant des transformations présentées ci-dessus appliquées à tous les points de S.

L'algorithme est alors le suivant :

Algorithm 1 Recherche de Nelder-Mead

Tant que (critère d'arrêt)
  Si $f(x_1) \leq min_i f(x_i^R)$
    Alors $S^* = S^C$
  Sinon si$f(x_1) > min_i f(x_i^R) > min_i f(x_i^E)$
    Alors $S^* = S^E$
  Sinon $S^* = S^R$
  finSi
finTq

Cet algorithme ne sera pas déterministe, car on se base sur une initialisation aléatoire uniforme dans l'espace des solutions pour la création du premier simplexe. La suite du calcul au contraire n'est aucunement stochastique. Le lancement s'effectue en mentionnant l'option *-m NMS* (*Nelder-Mead Search*).

4.5 Algorithme évolutionnaire

4.5.1 Principe

Les algorithmes évolutionnaires (dont les algorithmes dits "génétiques" sont une sous-famille), sont lointainement inspirés de l'évolution des populations, notamment du principe néo-darwiniste tel que la proportion d'individus les plus aptes à se reproduire (notamment grâce à leur survie, conséquente de leur adaptation au milieu, etc.) a tendance à s'accroître. Autrement dit, en considérant des individus mortels, le patrimoine génétique des plus fertile tend à se répandre parmi la population des générations suivantes. Dans notre problème on verra que tous les individus se reproduisent autant les uns les autres, mais que les couples ne sont pas choisis au hasard. Il ne se reproduiront pas à l'identique, mais comme les êtres vivants réels (mammifères, bactéries, etc.) formeront des couples pour créer des progénitures, dont le patrimoine génétique (soit la coordonnée sur chaque dimension) résultera d'un brassage des informations des deux parents. On peut alors présenter les mécanismes fondamentaux des algorithmes évolutionnaires :

- Sélection : un sous-ensemble de la population est choisi pour se reproduire, on forme les couples.
- Croisement : chaque couple donne naissance à un ou plusieurs enfants, dont les caractèristiques dépendent des parents.
- Mutation : une mutation peut se produire, indépendamment (en théorie) des parents.

On voit que la sélection permet d'identifier les meilleurs résultats, le croisement a pour but de les améliorer, et/ou de profiter de la bonne information de chacun pour créer un être ayant les qualité des deux parents, tout en restant dans le même axe de recherche, la mutation permet une exploration de l'espace de recherche, permettant par exemple de se sortir d'un optimum local. Grossièrement, ce sont les variantes de ces trois mécanismes, qui permettront d'obtenir des bons résultats. Décrivont maintenant l'algorithme type :

L'initialisation pourra être par exemple aléatoire uniforme dans l'espace de recherche. La mise à jour, quelque peu analogue à la sélection, consiste à réduire la taille de l'échantillon pour revenir à sa dimension initiale, en ne gardant que les meilleurs par exemple. Classiquement on applique un cruel élitisme, éliminant systématiquement les moins bons, mais on peut aussi appliquer d'autres mécanismes, citons par exemple la technique dite de la roulette, consistant à attribuer une probabilité de sélection proportionelle à la qualité de chaque individu (pour

Algorithm 2 Algorithme évolutionnaire

```
Initialisation
Tant que ( critère d'arrêt )
  Evaluation
  Sélection
  Croisement
  Mutation
  Mise à jour
finTq
```

nos minimisations la probabilité devrait être proportionelle à l'inverse de la valeur d'un point). La mutation est en général totalement stochastique, mais on peut imaginer toute sorte de mécanisme... ces types d'algorithmes vont du plus simple au plus sophistiqué, et l'efficacité n'est pas forcément proportionnelle à la complexité du calcul.

Les algorithmes de cette famille ne sont généralement pas réputés pour leur efficacité absolue, mais un de leurs interêts est qu'ils peuvent être modifiés à loisirs, et qu'on trouve des variantes des plus simples au plus exotiques, chaque algorithme étant en général propre à un problème donné, on ainsi verra par la suite qu'on ne peut pas forcément déclarer une variante systématiquement meilleure qu'une autre. Pendant mon stage j'ai pu assister à une journée de conférence (JET05) présentant diverses utilisation d'algorithmes évolutionnaires (ou apparentés) ; optimisation du partage de l'espace aérien, simulation de pousse de végetaux, évolution de produits financiers, etc.

On présentera un algorithme relativement simple, qui servira plus comme réference de comparaison que comme réel optimiseur, et un algorithme plus expérimental, que j'ai conçu à partir du simple, avec un certain nombre de modifications, dans l'idéal améliorations. Dans d'autres situations, pour suivre la métaphore biologique, on pourrait voir les gènes comme des bits ou séquences de bits, et ainsi effectuer les opérations de croisements et mutation sur des séquences de bits.

4.5.2 Mise en oeuvre

Une itération est divisée en trois pas : apprentissage (*learning*), diversification, intensification. A l'étape d'apprentissage, où on est muni de l'échantillon réduit (sans les enfants) de la génération précédente, on effectue le choix des parents et leur croisement pour fabriquer un certain nombre d'enfants. L'étape suivante, diversification, effectue la mutation (avec une certaine probabilité) sur chaque individu nouvellement crée (donc pas les parents), dans le but de l'améliorer, et finalement l'intensification réduit l'échantillon en ne conservant qu'une portion donnée de la famille. On voit donc que le but ultime de cette société n'est pas le

salut de l'individu mais le bien de la communauté, on pourra faire une analogie avec certaines populations d'insectes, ou certains états asiatiques.

4.5.3 Algorithme simple

Une version simple (simpliste) est implémentée, on l'utilise avec l'option *-m SGEN* (*Simple Genetic*). L'échantillon est initialisé selon une probabilité uniforme, autrement dit on tire chaque coordonnée au hasard dans les limites de l'espace donnée. On fait ensuite se reproduire tous les individus de la population, les meilleurs avec les meilleurs, les moins bons entre eux, chaque couple produisant deux enfants. On peut espèrer que deux individus avec des évaluations voisines se trouvent près l'un de l'autre dans l'espace de recherche, et que le déplacement conséquent du croisement sera minime.

4.5.3.1 Croisement

L'opérateur de croisement est défini de la façon suivante :

Soit f_i,m_i,c_i, $i = 1...n$, avec pour E l'espace de recherche, $dim(E) = n$, les coordonnées respectives, dans chaque dimension, du père, de la mère, et de l'enfant. Soit $R()$ un générateur aléatoire (en pratique pseudo-aléatoire) renvoyant une valeur rationelle au hasard (uniforme) entre 0 et 1 . On calcule les $c_i$de la façon suivante :

Algorithm 3 Croisement simple

Initialisation du vecteur cà la dimension n
Pour chaque dimension i :
  $P = R()$
  $c_i = P \times f_i + (1 - P) \times m_i$
finPour

Ainsi la somme des coordonnées des deux parents est normalisée, et on aura une influence aléatoire de l'un et de l'autre. On est sûr (pour le meilleur et pour le pire) qu'on aura :

$min(f_i, m_i) \leq c_i \leq max(f_i, m_i)$

On ne sortira pas donc de l'hypercube défini par les extrema sur chaque dimension de chaque point de l'échantillon, même sur plusieurs générations, sauf si on opère des mutations.

4.5.3.2 Mutation

Cet algorithme se voulant simple, on opère la mutation de la façon suivante :

Cette mutation, aussi simpliste soit-elle, permettra d'explorer l'espace de recherche.

Algorithm 4 Mutation simple

$P = R()$
Si $P < mutationProba$, alors
  Réinitialiser toutes les coordonnées du point aléatoirement
Sinon
  Ne rien faire
finSi

4.5.3.3 Sélection

La sélection finale se fait en sélectionnant les $N * selectionCoef$ meilleurs, où N est la taille de l'échantillon fixée par le problème, et $selectionCoef$ un coefficient entre 0 et 1, fixant la proportion d'individus à sélectionner, on fixe cette valeur à 0.5, ainsi pour un échantillon de 10 individus, on en sélectionnera finalement 5, qu'on fera ensuite se reproduire pour atteindre une population totale de 10 individus.

4.5.4 Algorithme expérimental

4.5.4.1 Croisement

Un des problèmes supposés de l'algorithme simple concerne l'extension de l'hypercube de l'échantillon : cet hypercube sera ici la figure à N dimensions, N étant le nombre de dimensions de l'espace de recherche, formé par les extremas (minimum et maximum) des points de l'échantillon sur chaque dimension. En effet le résultat du croisement est un point dont chaque coordonnée est une combinaison linéaire normalisée des coordonnées des parents tel que la nouvelle coordonnée soit plus ou moins proche de l'un ou de l'autre, mais reste forcément entre les deux. La mutation a pour but d'explorer l'extérieur de l'hypercube, mais cette exploration n'est pas du tout intelligente, au sens où d'une part on n'est pas sûr du tout de tirer un point hors de l'hypercube, et d'autre part on a encore moins de chance de le tirer dans la zone vers laquelle on aimerait s'orienter, par exemple dans le voisinage de l'optimum courant, mais hors de l'hypercube. Pour résoudre ce problème on modifiera les fonctions de mutation et de croisement.

Initialement le croisement se faisait de la même façon que pour l'algorithme simple, comme une combinaison linéaire normalisée des coordonnées des parents, tel que :

$c_i = P \times f_i + (1 - P) \times m_i \, , \, 1 \leq i \leq N$

Seulement on se rend vite compte que cette méthode n'a pas vraiment de sens, et à partir de deux "bons" points, on peut obtenir le pire, pour celà prenons un exemple, en deux dimensions. Soit les points parents $F(0.1, 0.1)$ et $M(0.8, 0.8)$, sur un espace ce recherche $(0..1, 0..1)$. On voit que ces deux points sont très

éloignés, et que leur bonne valeur ne garantie aucunement qu'un point dans leur hypercube soit acceptable. On peut obtenir par exemple le point $C(0.7, 0.2)$, qui, s'il approche le minimum, sera seulement le fruit d'un heureux hasard. Par exemple si l'optimum réel se trouve entre l'origine et F, on n'aura aucune chance de l'approcher.

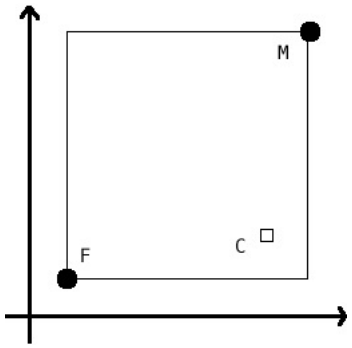


FIG. 4.1 – Positions du point C, résultat du croisement de Met F.

A fortiori, dans un espace à 30 dimensions, pour deux points éloignés, le résultat aura peut de chances d'être satisfaisant. Ce croisement équivalent à un tirage aléatoire dans l'hypercube définit par les deux parents n'est donc pas intéressant dès que les parents sont trop éloignés. Même en N dimensions, il suffit qu'une des coordonnées de l'optimum se trouve hors des limites définies par les parents pour qu'on soit certain de ne pas l'approcher. On a donc essayé une solution alternative, dont le principe est d'explorer le voisinage d'un des deux points, en fonction des coordonnées de l'autre. Et on peut aisément prévoir l'inefficacité de la fonction de croisement simple pour des problèmes avec un grand nombre de dimensions.

La nouvelle solution partait du principe suivant : chaque couple de points, un mâle et une femelle (on sexualise les points pour les différencier l'un de l'autre, celà n'implique aucune différence effective), produisent deux enfants, un garçon et une fille (*idem*). Et les coordonnées de chaque enfant seront tirées à partir d'un décalage de la coordonnée du parent du même sexe, en fonction aussi d'un nombre aléatoire et de la coordonnée de l'autre parent. Malheureusement, les résultats furent assez décevants : on avait d'abord implémenté le nouveau modèle de mutation, qui avait démontré une bien meilleure efficacité de l'algorithme par rapport au simple, sur certains problèmes. Une autre idée est d'orienter la position de l'enfant, toujours en fonction de son sexe, mais aussi vers le parent ayant la meilleure évaluation. Formellement, on accordera un coefficient d'autant plus grand à un parent que son évaluation est bonne (en normalisant la somme des coefficients). A priori ceci devrait orienter l'algorithme vers la meilleure zone, mais on constate une fois de plus que l'expérience contredit l'intuition :

on obtient de meilleurs résultats avec la version de base du croisement !

Finalement, on conservera la version simple de la fonction de croisement, qui offre une meilleure diversité que les tentatives vaines de variantes, au prix d'une efficacité théorique plus que douteuse. On va alors constater l'important de la mutation, qui sera décisive pour l'efficacité de notre algorithme. Pour le choix des couples on utilisera toujours la méthode consistant à sélectionner les paires dans l'ordre des valeurs des points, les meilleurs se reproduisants avec les meilleurs, les moins bons avec les moins bons. L'alternative d'effectuer un tirage aléatoire dans l'échantillon a été testé ; on ne constate pas de gain de performance, c'est au contraire légèrement moins satisfaisant. De plus cette méthode présente l'inconvénient a priori de ne pas forcément utiliser tous les points de l'échantillon, et d'avoir des redondances si certains participent à plusieurs reproductions. Celà dit on pourrait programmer l'algorithme de telle façon qu'un individu ne se reproduise pas plusieurs fois, comme c'est le cas pour la version de base.

4.5.4.2 Mutation

Le principe de la mutation est le suivant : on a une probabilité de mutation donnée, la même pour chaque point (on pourrait envisager une probabilité dépendant de la position ou valeur du point). On itère sur chaque dimension le mécanisme, avec la même probabilité. Si une mutation effective doit alors avoir lieu (dans le cas échéant, les solutions du point ne sont pas modifiées), une seconde probabilité intervient, celle d'avoir une mutation totale, ou partielle. La mutation totale reviendra, comme pour l'algorithme simple, à effectuer un tirage aléatoire dans l'espace des solutions. Pour la mutation partielle, on aura alors trois possibilités :

- Accroissement (In)
- Diminution (De)
- Rien (No)

L'accroissement et la diminution correspondent au produit de la coordonnée par un coefficient donné, entre 0 et 1, soit par son inverse (respectivement pour diminution et accroissement). Nous verrons par la suite qu'un autre coefficient devra intervenir. N'ayant pas d'indice a priori pour orienter la recherche, on fixe une probabilité identique pour chacun de ces évènements :

$P(In) = P(De) = P(No) = \frac{1}{card\{In,De,No\}} = \frac{1}{3}$

On a toujours le générateur aléatoire $R()$, qui tire un nombre rationnel entre 0 et 1, selon une loi de probabilité uniforme.

La mutation est caractèrisée par trois valeurs :

- $mutationProba$ est la probabilité d'une mutation.
- $totalMutationProba$ celle d'une mutation totale, sachant qu'on doit faire une mutation, totale ou partielle.

– *reduction* est le taux de réduction après application du coefficient, lors d'une mutation partielle (dans $[0..1] \subset \mathbb{Q}$).

Les $p_i$sont les coordonnées du point pour chaque dimension i.

Algorithm 5 Mutation expérimentale

Si la valeur de p est supérieure (moins bonne) à la meilleure valeur courante, alors
  $P = R() \times mutationProbe$ (on force la mutation)
Sinon (probabilité $mutationProba$ de mutation)
  $P = R()$
Si $P < mutationProba$, alors
  Si $P < mutationProba \times totalMutationProba$, alors (*mutation totale*)
    Initialisation aléatoire de chaque coordonnée p_i
  Sinon (*mutation partielle*)
    Pour chaque dimension i
      $P = R()$
      $Coef = R()$
      Si $P < \frac{1}{3}$, alors (*diminution*)
        $p_i = p_i \times Coef \times reduction$
      Sinon si $P < \frac{2}{3}$, alors (*accroissement*)
        $p_i = p_i \times (1 - Coef) \times reduction$
      Sinon (*ne rien faire*)
        $p_i = p_i$
      finSi
    finPour
  finSi
finSi

On vérifiera pour l'accroissement et la diminution que la nouvelle coordonnée se trouve bien dans les limites de l'espace de recherche. Si tel n'est pas le cas, on laisse la coordonnée telle quelle. L'avantage de cette méthode de mutation est la capacité qu'elle offre aux enfants de s'extraire de l'hypercube définit par les parents, tout en conservant tout ou partie de leur héritage, et une certaine proximité des parents, grâce au facteur *reduction* définissant en quelque sorte le rayon maximal de l'hypersphere centrée sur le point considéré (avant mutation partielle) dans laquelle on trouvera le point muté, ainsi fixer *reduction* à l'unité revient à tirer la nouvelle coordonnée dans tout l'espace de recherche (la loi de probabilité n'étant alors plus uniforme, mais d'autant plus décalée que l'est le point dans la dimension considérée). Ceci sera décisif pour la recherche sur les "bords" de l'espace de recherche, et permettra d'équilibrer une optimisation des solutions des parents avec une exploration locale.

Une optimisation théorique pourrait consister à définir la probabilité $totalMutationProba$ dynamiquement, de façon à inciter l'exploration (donc les mutations totales) au début de l'algorithme, et ensuite les réduire pour favoriser la recherche locale.

Pour celà on doit connaître la durée du calcul, soit de façon discrète le nombre d'évaluation (ou d'itérations), et par exemple définir une évolution linéaire où la valeur minimale de $totalMutationProba$ serait atteinte avant la fin du calcul. Il serait même préférable que la réduction soit assez brusque vers la fin, car l'impact d'une faible probabilité serait quasi-nulle, surtout si on se trouve dans un espace à plus de dix dimensions, où la probabilité de tirer un point dans un zone (encore plus) favorable relève de l'intervention divine. Mais une fois la régression linéaire implémentée, on constate une fois de plus que l'effet n'est pas celui attendu, et que les résultats sont légèrements moins bons, on conservera finalement la probabilité statique.

N'ayant pas d'approche théorique suffisament poussée pour prédire la meilleure valeur des trois paramètres de la mutation ($mutationProba$, $totalMutationProba$, et $reduction$), on les fixe d'abord à l'intuition, et après quantité de tests, comparaisons sur différents problèmes et ensembles de paramètres, on peut considérer que les valeurs suivantes sont relativement efficaces (on pourrait méta-optimiser en utilisant une métaheuristique pour déterminer la meilleure solution, qui se situerait alors ici dans un espace à trois dimensions) :

- $mutationProba = 0.7$: chaque enfant a donc sept chances sur dix de muter, autrement dit de voir ses coordonnées modifiées, totalement ou partiellement.
- $totalMutationProba = 0.4$: si une mutation a lieu, c'est la probabilité que celle-ci soit totale, on aura alors une probabilité non conditionnelle de $0.4 \times 0.7 = 0.28$.
- $reduction = 0.5$: la réduction du coefficient de mutation partielle, pour ne pas trop éloigner le point de sa position d'origine.

On utilisera cet algorithme en mentionnant l'option *-m JGEN* (*Jp's Genetic*).

4.6 Comparaison de performances

Nous présenterons les graphiques les plus significatifs pour deux ensembles de tests, afin de comparer les efficacité relatives des algorithmes implémentés. Le lecteur est invité à tester lui-même le programme, par exemple avec les scripts de *demo.py*, pour étudier toutes les données que nous ne pourrons détailler ici, faute de place.

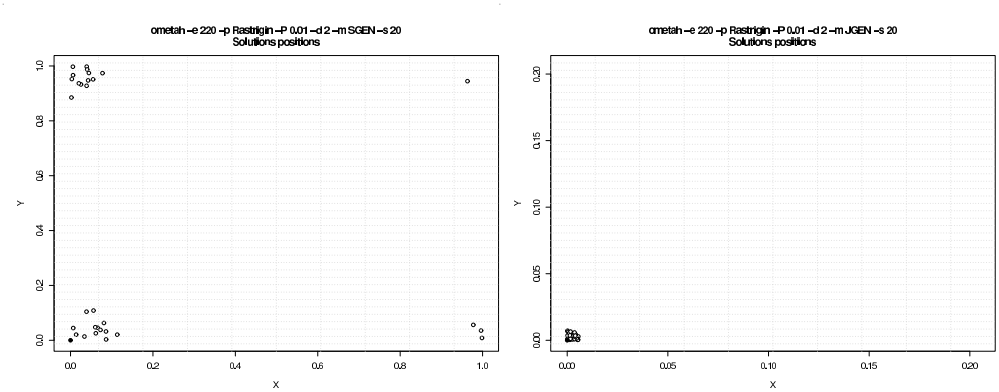
4.6.1 Problème Rastrigin à deux dimensions

On a déjà constaté (voir 2.3.2) l'allure bossellée de cette fonction multimodale. On étudie les tests suivants, chacun avec un algorithme différent, identifiés par leur paramètres (ligne de commande de ometah) :

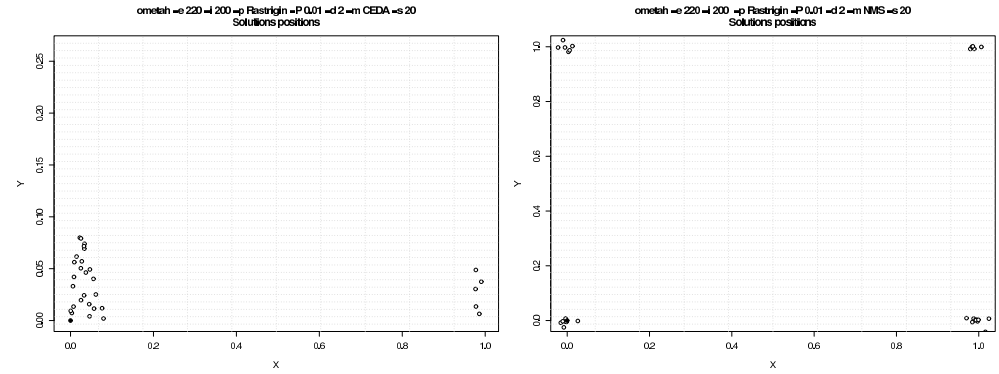
1. *-e 220 -p Rastrigin -P 0.01 -d 2 -m SGEN -s 20*
2. *-e 220 -p Rastrigin -P 0.01 -d 2 -m JGEN -s 20*

3. *-e 220 -p Rastrigin -P 0.01 -d 2 -m CEDA -s 20*
4. *-e 220 -p Rastrigin -P 0.01 -d 2 -m NMS -s 20*

Pour qui n'est pas habitué à ces notations, on lance simplement ometah sur le problème Rastrigin à deux dimensions, avec un échantillon de 20 points, le critère d'arrêt étant soit le maximum d'évaluations fixés à 220, soit une précision de 0.01 par rapport à l'optimum (situé en $0,0$ pour ce problème), et on précise dans le script que chaque test effectuera 30 runs. On identifiera désormais ces tests par leur index, de 1 à 4, comme c'est le cas dans les graphes. On observe d'abord les positions des optima de chaque runs, l'optimum étant représenté par le point rempli, sur le graphe de l'espace des solutions (figure 4.2).



(a) SGEN (b) JGEN



(c) CEDA (d) NMS

FIG. 4.2 – Positions des optima dans l'espace pour Rastrigin, 2 dimensions.

Ces graphes informent seulement sur les positions des points dans l'espace (à deux dimensions) ici, et non sur les valeurs, mais on peut tenter d'en inférer une approximation des efficacités relatives des algorithmes ; l'algorithme évolutionnaire simple (a) semble a priori le moins bon, on distingue clairement les optima locaux (les cavités du champ de bosses), à chaque coin de l'espace de recherche. Dans la bonne cavité, celle de l'optima, on a plus de points que dans les autres, mais ce n'est pas flagrant. L'algorithme évolutionnaire modifié ne se perd pas dans les optima locaux de ce problème, et on observe une forte densité dans le voisinage de l'optimum, on peut donc supposer qu'une bonne valeur ai été obtenue. L'estimation de distribution se démarque des autres algorithmes : la topologie de l'espace est moins marquée, et la plupart des points se trouvent dans la zone de l'optima, mais assez dispersés. Quant à la recherche de Nelder-Mead, elle semble assez efficace localement (forte densité pour chaque optimum local), mais peine à se concentrer sur le minimum global.

Observons maintenant le graphe des valeurs, pour chacun de ces tests, où on affiche pour chaque test (en abscisse), le meilleur optima des 30 runs, le pire, et la valeur mediane sur l'ensemble des runs (figure 4.3). Les hypothèses réalisées à partir des positions des points sont confirmées, l'algorithme évolutionnaire simple obtient les moins bons résultats, la version modifiée les meilleurs, et entre les deux l'estimation de distribution et la recherche de Nelder-Mead. De plus l'écart est particulièrement important, sachant que les différences entre les deux algorithmes évolutionnaires ne sont pas si grandes que ça, on peut alors mesurer l'impact d'une mutation optimisée.

On n'affichera que les convergences des optima pour les deux meilleurs algorithmes, qui révèlent une plus grande variance des valeurs dans l'échantillon pour la recherche de Nelder-Mead, mais une valeur médiane relativement statique. A l'opposé l'algorithme évolutionnaire présente une assez grande densité dans les valeurs, et les variations sont bien plus brusques, car il suffit qu'une mutation totale ou partielle révèle une bonne zone pour que celle-ci prenne l'ascendant à la génération (itération) suivante. Donc l'impact de l'initialisation aléatoire est plus important sur NMS (grande variance entre les runs), mais celui de l'exploration aléatoire l'est plus sur JGEN, où les runs tendent naturellement à sélectionner les mêmes (les meilleurs) zones de l'espace.

4.6.2 Problème Rosenbrock à dix dimensions

Comme précédemment on présentera d'abord la répartition des points dans l'espace, puis la comparaison des valeurs obtenues. Mais tout d'abord présentons les tests effectués :

1. *-e 500 -i 200 -p Rosenbrock -P 0.01 -d 10 -m SGEN -s 20*
2. *-e 500 -i 200 -p Rosenbrock -P 0.01 -d 10 -m JGEN -s 20*
3. *-e 500 -i 200 -p Rosenbrock -P 0.01 -d 10 -m CEDA -s 20*

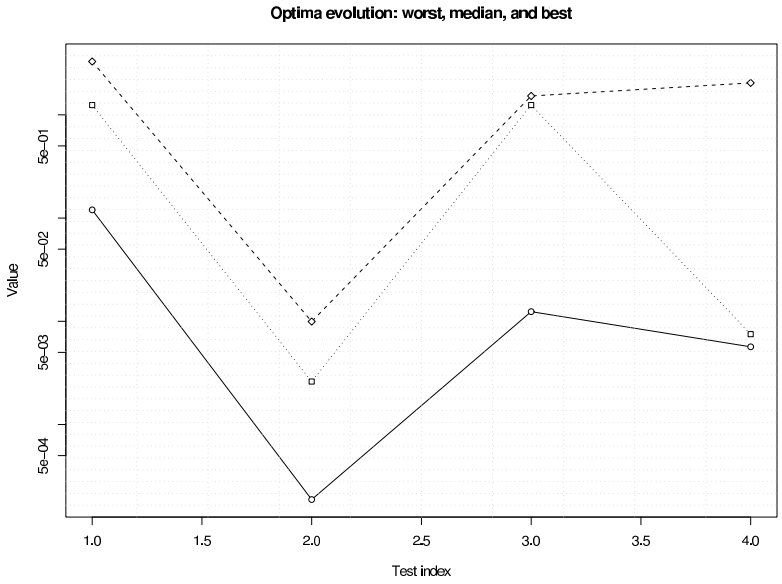
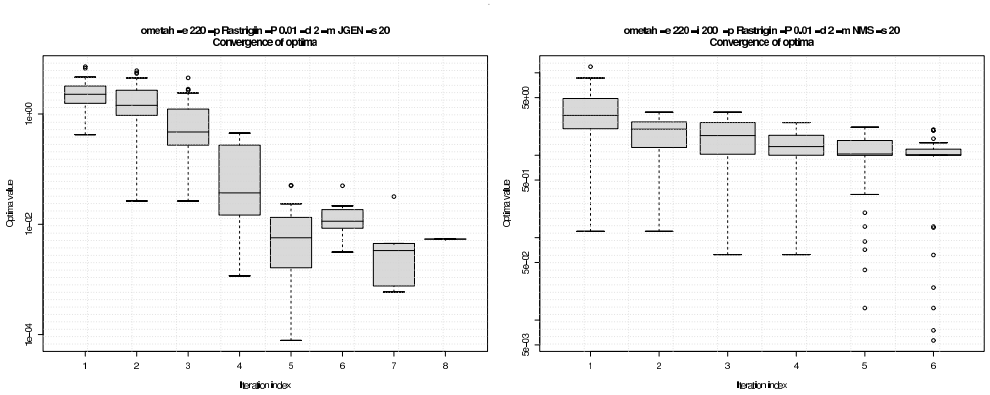


FIG. 4.3 – Graphe des valeurs pour Rastrigin à 2 dimensions.

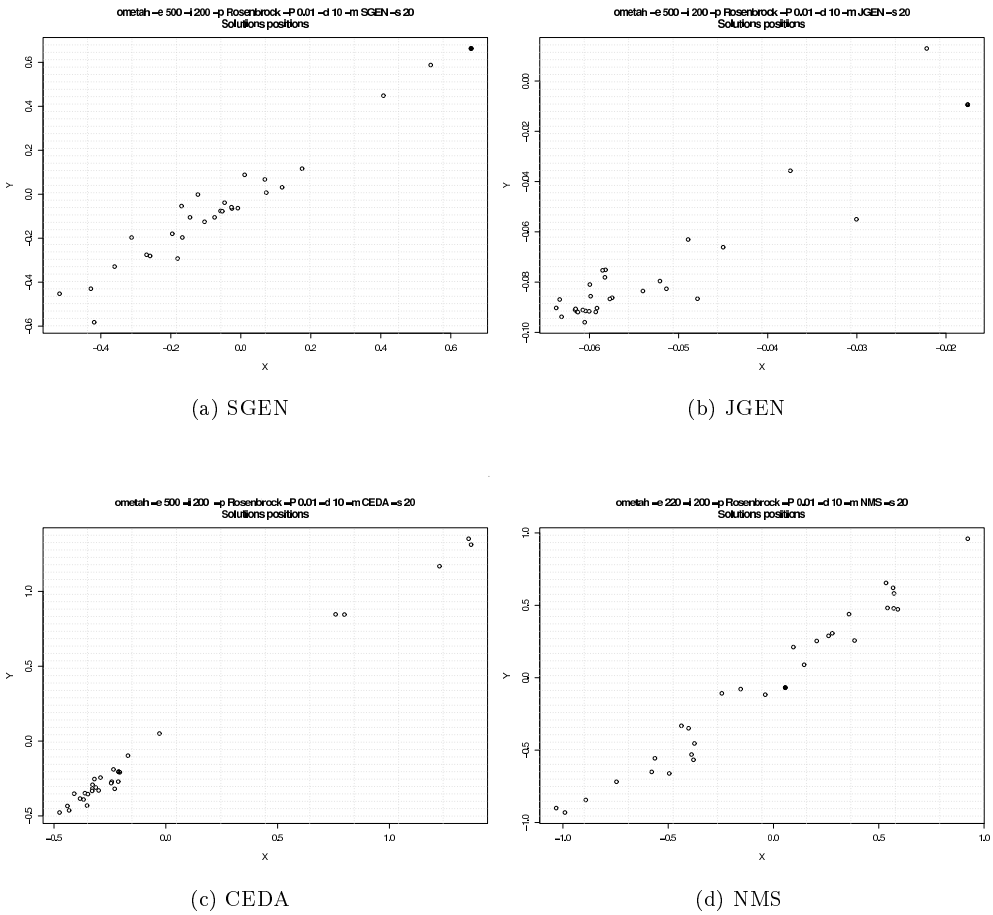
4. *-e 500 -i 200 -p Rosenbrock -P 0.01 -d 10 -m NMS -s 20*

Le problème sera donc particulièrement difficile, d'une part Rosenbrock est multimodal et présente plusieurs optima locaux dans la zone de l'optimum, et les dix dimensions accroissent évidemment la difficulté. On considère un maximum de 500 évaluations, en indiquant *-i 200* comme majorant, pour être certain que la valeur par défaut du nombre d'itérations maximal n'interrompe pas le calcul avant les 500 évaluations.



(a) JGEN (b) NMS

FIG. 4.4 – Convergences des optima.



(a) SGEN (b) JGEN

(c) CEDA (d) NMS

FIG. 4.5 – Position des optima dans l'espace pour Rosenbrock, 10 dimensions.

A la vue des positions des points (figure 4.5), on placerait a priori l'estimation de distribution en tête, suivie de NMS, SGEN, et JGEN. Pour ce dernier la concentration de points est excentrée de la zone de l'optimum, on pense alors à un minimum local. Cependant on doit également regarder l'échelle, qui varie à chaque pas d'un dixième, sauf pour JGEN, où on varie d'un centième, et donc où on a une meilleure précision, mais la présence d'un optimum local très proche de l'optimum réel fausse cependant le résultat. L'affichage des valeurs confirme alors ces différences supposées (figure 4.6). Notons que l'affichage des positions des points se fait à partir d'une réduction par analyse en composantes principales, d'un espace à dix dimensions en un espace à deux dimensions. Comme les poids des axes tendent à être équivalents, on retrouve la forme de la vallée qu'on trouverait dans l'affichage des positions du problème à deux dimensions seulement.

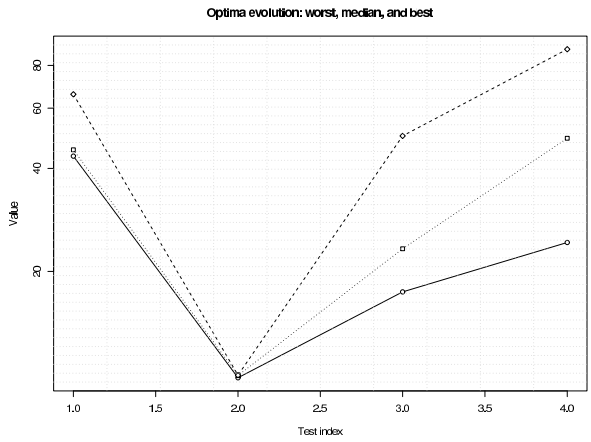


FIG. 4.6 – Graphe des valeurs pour Rosenbrock à 10 dimensions.

Ainsi l'algorithme évolutionnaire semble relativement efficace, cependant on sait que les métaheuristiques ne sont pas forcément universellement meilleures ou moins bonnes que d'autres, et on constate que pour des dimensions inférieures, et ce seulement sur le problème Rosenbrock, il produit d'assez mauvais résultats, a priori en restant bloqué dans un optimum local de valeur moyenne (figure 4.7). Ceci semble assez étrange a priori, car en dix dimensions les résultats sont toujours bons, alors que la difficulté est plus grande. Qui peut le plus ne peut donc pas forcément le moins.

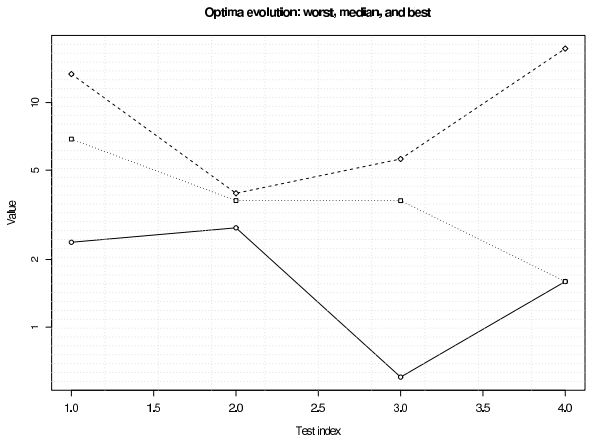


FIG. 4.7 – Graphe des valeurs pour Rosenbrock à 4 dimensions.

4.7 Problème Sphere à cinquante dimensions

On se limite cette fois aux deux algorithmes ayant montré les meilleures performances, pour s'attaquer à un problème relativement simple, l'hyperparabolle Sphere, unimodal, mais à cinquante dimensions, ce qui corse un peu la difficulté. On considère cette fois un échantillon de dix points seulement, et le plus grand nombre d'itération permettra d'observer une convergence sur une plus longue période. Ici les positions dans l'espace apportent peu d'interêt visuellement (la réduction de cinquante à deux dimensions élimine une grande partie de l'information), mais on observe une répartition des points dans le voisinage de l'optimum pour chacun des algorithmes, où la distance moyenne est de l'ordre de $1e-4$ pour JGEN, et du dixième pour NMS. Précisons tout de même les lignes de commande invoquées :

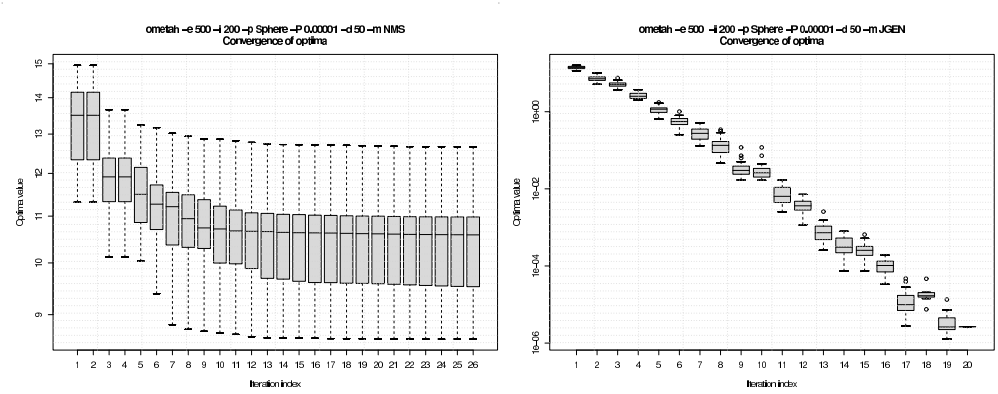
1. *-e 500 -i 200 -p Sphere -P 0.00001 -d 50 -m NMS*
2. *-e 500 -i 200 -p Sphere -P 0.00001 -d 50 -m JGEN*

Les valeurs des optima obtenus sont les suivantes :

1. NMS : 8.562
2. JGEN : 1.266e-06

Il est ici plus intéressant d'étudier la convergence des échantillons, pour voir l'allure des courbes de progression (attention à l'échelle logarithmique !). On constate encore que les échantillons de NMS présentent une plus grande variance, pour converger rapidement et ne plus améliorer l'optimum, pourtant il

y'aurait lieu, la valeur obtenue est relativement grande, et aucun optimum local n'est présent pour cette fonction. Ce comportement peut-être du au caractère déterministe de cette technique de recherche. A l'opposé JGEN présente des échantillons de faible variance, mais chacun étant très différent du précédent, et atteint ainsi une bonne précision à la fin du calcul, et on peut supposer qu'il poursuivrait le raffinement si on avait plus d'évaluations.



(a) NMS

(b) JGEN

FIG. 4.8 – Convergences des optimas pour Sphere à 50 dimensions.

4.8 Rosenbrock à 50 dimensions

Avec d'autres algorithmes récemment implémentés, on s'attaque au problème Rosenbrock, probablement le plus difficile, à 50 dimensions, avec les lignes de commande suivantes :

1. *-e 5000 -p Rosenbrock -P 0.01 -d 50 -m CEDA -s 10*
2. *-e 5000 -p Rosenbrock -P 0.01 -d 50 -m CHEDA -s 10*
3. *-e 5000 -p Rosenbrock -P 0.01 -d 50 -m HCIAC -s 10*
4. *-e 5000 -p Rosenbrock -P 0.01 -d 50 -m JGEN -s 10*
5. *-e 5000 -p Rosenbrock -P 0.01 -d 50 -m NMS -s 10*
6. *-e 5000 -p Rosenbrock -P 0.01 -d 50 -m SGEN -s 10*

HCIAC est un algorithme à colonies de fourmis, CHEDA un hybride de CEDA et NMS. On obtient le graphe des valeurs suivant :

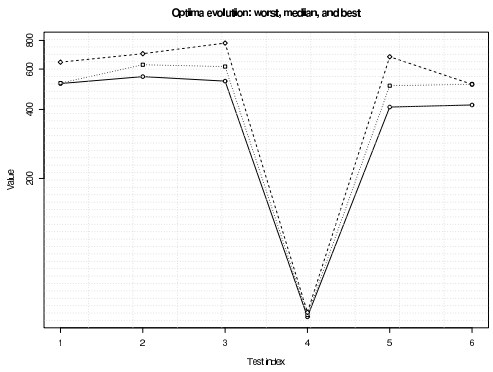


FIG. 4.9 – Graphe des valeurs pour Rosenbrock à 50 dimensions.

Troisième partie

Conclusion

4.9 Evolution du logiciel

Dans l'état actuel les modules de ometahlab sont suffisament fonctionnels pour effectuer des tests sur les problèmes et algorithmes fournis par ometah, le logiciel a été testé pour de plus lourds calcul, et semble désormais exempt de bugs trop récurrents. Il reste à découvrir tous les autres, et les corriger. De plus une utilisation fréquente et réellement expérimentale amènera sûrement à constater des lacunes ou erreurs, et le programme sera de toutes façons amené à être modifié. Il pourra être utilisé (et modifié) en premier lieu par les chercheurs du LISSI travaillant sur l'optimisation, et pourquoi pas par d'autres universités, mais le relais pour la maintenance sera certainement assuré par mon maître de stage, qui a suivi le développement du logiciel, et est le plus apte à comprendre aisément son fonctionnement, donc à en détecter les erreurs. Je serai toujours disponible pour expliquer ou corriger des portions de codes trop obscures, justifier tel ou tel choix, et éventuellement rajouter ou corriger des fonctionnalités. De plus, ometahlab est indissociable du programme C++ ometah, dont j'ai réalisé l'interface et programmé quelques algorithmes, et l'évolution du second impliquera probablement des modifications sur le premier. Ainsi donc j'aurais accompli la mission qui m'était fixée, répondant à un besoin, en participant à un logiciel qui pourra aider les chercheurs de ce domaine, en leur faisant ainsi économiser un nombre d'heures considérables qui auraient été consacrées à l'implémentation d'un programme informatique. Le statut de logiciel libre permettra la mise en commun du code du programme, manifeste du principe de partage des connaissances, si utile à la communauté scientifique.

Du point de vue technique, le logiciel étant relativement jeune (ometahlab) pourra pêcher par son instabilité, et on peut prévoir des comportements inattendus lors de situations singulières ou extrêmes, par exemple si le nombre d'évaluations est très élevé. Les tests réalisés ont permis de corriger certaines erreurs, mais étant donné la durée de calcul d'un ensemble de tests lourds on n'a pu se permettre de trop les multiplier. Une possibilité pour la suite serait de réaliser un système de calcul parallèlisé sur plusieurs machines en réseau pour accélerer les tests. Une fois Open Metaheuristic parvenu à un état stable, il pourrait être utile de rédiger une meilleure documentation, portant sur chacun des modules (ometah et ometahlab), et de mettre le logiciel à disposition du public sous forme de binaires, packages, etc., pour inciter à son utilisation.

4.10 Apport personnel

Ces deux mois passés à travailler au sein d'un laboratoire de recherche universitaire m'ont permis de confronter mes a priori avec la réalité, et d'en savoir un peu plus sur la condition de chercheur et de doctorant dans un établissement public. La majorité des chercheurs n'étant pas de spécialité informatique, mais de culture plus scientifique (physique, biologie), ce fut également l'occasion de découvrir un autre milieu, et notamment d'en étudier le rapport avec

l'informatique (besoins, logiciels, compétences), et les informaticiens (des programmeurs). J'ai également bénéficié d'échanges enrichissant avec certains doctorants, qui m'ont notamment mis en garde une fois de plus sur les épreuves de toutes sortes auxquelles est confronté un thésard.

Ce stage à tout à fait répondu à mes attentes, et m'a été très bénéfique aussi bien techniquement que scientifiquement : j'ai utilisé des langages auxquels je n'étais pas habitué (C++, Python), dont je possède désormais une certaine maîtrise, et appris à (mieux) manipuler certains logiciels que j'ai de fortes chances d'utiliser dans la suite de mon cursus (CVS, R, Lyx, etc.). Le double aspect du sujet, à la fois théorique et expérimental, m'a été très enrichissant ; j'ai pu découvrir un domaine de l'optimisation qui m'était inconnu, en étudiant les différentes métaheuristiques et leurs applications, que j'ai ensuite pu tester grâce au logiciel que j'ai créé. J'ai aussi appris quelques principes élémentaires à respecter dans le cadre d'une démarche scientifique expérimentale, concernant la rigueur à adopter pour la présentation des résultats et leur interprétation, et les outils statistiques à utiliser, ceci faisant souvent défaut aux publications scientifiques d'informaticiens. La mise en valeur de résultats se doit d'être effectuée judicieusement, il est par exemple facile de démontrer l'efficacité de n'importe quel algorithme en comparant ses performances avec celles d'un autre moins bon, en disant juste "Cet algorithme donne de très bons résultats !". Dans la continuité de l'implémentation d'algorithmes j'ai pu en réaliser une variante originale, et démontrer ses bonnes performances par rapport aux autres méthodes du programme. Toutefois il convient de ne pas trop se féliciter, car d'autres algorithmes plus complexes sont certainement bien plus performant, et l'optimisation des paramètres que j'ai réalisée pour mon algorithme n'a pas été faite de la même manière pour les autres.

Enfin, j'ai pu constater l'autonomie et la liberté offerte aux chercheurs, ses bienfaits et méfaits, me rendre compte "de l'intérieur" des difficultés auxquelles auxquelles sont confrontés certains jeunes chercheurs, mon jugement actuel contrastant avec l'idéalisation naïve passée ne pourra désormais que m'inciter à faire de mon mieux pour la suite.

Quatrième partie

Annexe

Formules et graphes des problèmes

Les fonctions Schwefel, Sphere, Rastrigin et Rosenbrock ont déjà été vues au début de ce rapport, nous présenterons ici les autres problèmes disponibles dans Open Metaheuristics, ce qui permettra d'effectuer des tests en connaissance de cause, et de mieux en interpréter les résultats (notamment pour situer des optima locaux, et estimer la difficulter du problème). Tous ces problèmes sont multimodaux (ils possèdent plusieurs extrema locaux).

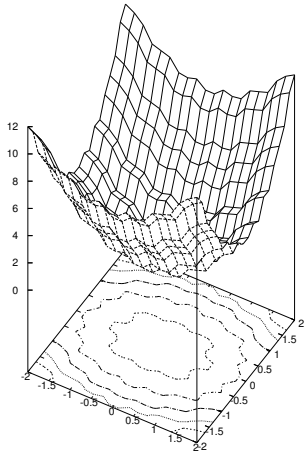
Ackley $F_5(x) = -20e^{\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2}\right)} - e^{\left(\frac{1}{D}\sum_{i=1}^{D} cos(2\pi x_i)+20+e\right)}$

Griewank $F_6(x) = \sum_{i=1}^{D}\left(\frac{x_i^2}{4000}\right) - \prod_{i=1}^{D}\left(\cos\left(\frac{x_i}{\sqrt{i}}\right)+1\right)$

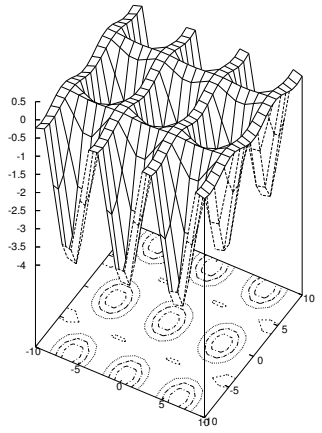
Weierstrass $F_7(x) = \sum_{i=1}^{D}\left(\sum_{k=0}^{k_{max}}(a^k\cos(2\pi b^k(x_i+0.5)))\right) - D\sum_{k=0}^{k_{max}}(a^k\cos(2\pi b^k\cdot 0.5))$

Exemple de problème non-continu (non-intégré à ometah) On utilise la fonction $E(x)$qui renvoie la partie entière de x. Cette fonction présente donc des discontinuités, et on aura une infinité d'optima (si les coordonnées ne sont pas discrètes), soit tous les points du plan le plus bas.

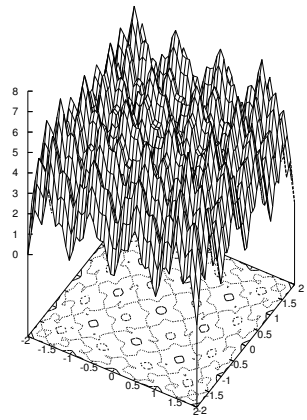
$F_s(x) = 25 - (\sum_{i=1}^{D} E(x_i+1))$



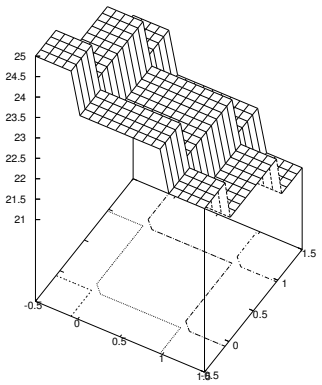
(a) Fonction Ackley.



(b) Fonction Griewank.



(c) Fonction Weierstrass.



(d) Fonction discontinue.

FIG. 4.10 – Représentations des problèmes en trois dimensions.

Portions de code

Ayant écrit durant ce stage au moins 5000 lignes de code, il m'a semblé utile de présenter un minimum d'extraits des divers programmes produits, notamment pour vanter les mérites de Python et démystifier certaines opérations facilement réalisées grâce aux nombreux modules importés. J'ai utilisé Emacs pour chaque source créé, ses nombreux raccourcis et son adaptation à chaque langage ont certainement beaucoup compté dans la rapidité du développement. Sachant que la lecture de code est souvent rébarbative, que deux ou trois bits suffiront à coder en binaire le nombre de mes lecteurs, et que j'aimerais tout de même limiter le nombre de pages de ce rapport, je tâcherai d'être bref.

Procédure pour la génération d'un graphique, en l'occurence celui des boîtes de convervence des points, on identifiera facilement le rôle de chaque instruction :

```
def __plot_4(self) :
""" For each Test, plot the sequence of iterations (for any run) as a set of quantile boxes. So we have one graphic for each Test. """
fileName = os.path.join(self.__dir, 'convergence_points.ps')
r.postscript(fileName, paper='letter')
i = 0
for metalist in self.__pointsIter :
vlist = [[p.value for p in points] for points in metalist ]
txt = '%s\nConvergence of all points' % self.__tests[i].args
r.boxplot(vlist, style='quantile', col=self.__color, log="y", main=txt, xlab='Iteration index', ylab='Point value')
r.grid(nx=10, ny=40)
i += 1
r.dev_off()
```

La définition des classes Python est relativement (à C++ notamment) simple, *__init__* est le constructeur, et chaque méthode de la classe devra avoir comme premier paramètre *self*, soit l'objet lui-même, passé implicitement à l'appel, pour signifier à quelle instance appliquer la méthode. On présente ici les classes *Point* et *Problem*, avec leurs attributs naturels (le *self.*attribut signifie que c'est un attribut de classe) :

```
class Point :
```

```
""" A point has a set of coordinates, a value, an error relative to the problem's
optimum, and an index. """

  def __init__(self) :
""" Void constructor."""
self.coords = None
self.value = None
self.error = None
self.index = None

class Problem :
""" Descriptive informations of a problem. """

  def __init__(self) :

    """ Void constructor."""
self.key = None
self.name = None
self.description = None
self.formula = None
self.dimension = None
self.optimum = [] # list of Point
self.min_bound = [] # idem
self.max_bound = [] # idem
self.reference = None
self.accuracy = 0.0
```

Et enfin un extrait de code C++, l'implémentation de l'algorithme de croisement des algorithmes évolutionnaires (le même pour les deux) :

```
vector<itsPoint> itsJpGenetic : :makeChildren(itsPoint father, itsPoint mo-
ther) {
double alpha;
itsPoint boy;
itsPoint girl;

  // solutions vectors for father, mother, boy and girl
vector<double> fsol, msol, bsol, gsol;

  fsol = father.getSolution();
msol = mother.getSolution();

  // for each dimension, set the coordinate, ~ find a point with uniform proba
in hypercube of parents coords.
for (int i=0; i<this->problem->getDimension(); i++) {
alpha = drand48(); bsol.push_back( fsol[i] * alpha + msol[i] * (1 - alpha) );
alpha = drand48(); gsol.push_back( fsol[i] * alpha + msol[i] * (1 - alpha) );
}

  boy.setSolution( bsol );
girl.setSolution( gsol );
```

```
vector<itsPoint> v;
v.push_back( evaluate(boy) );
v.push_back( evaluate(girl) );

  return v;
}
```

Bibliographie

[1] M.Bierlaire, *Algorithmes de recherche directe*, cours d'optimisation de l'EPFL.

[2] M.Clerc, *Semi-continuous challenge*, 2004, *http ://clerc.maurice.free.fr*.

[3] J.Dréo, A.Petrowki, P.Siarry, E.Taillard, *Métaheuristiques pour l'optimisation difficile*, Eyrolles, 2003.

[4] J.Dréo, thèse *Adaptation de la méthode des colonies de fourmis pour l'optimisation en variables continues*, 2004.

[5] P. Laragana, J.A. Lozano, *Estimation of distribution algorithms*, 2002.

[6] P.N. Suganthan et al, *Problem definitions and evaluation criterias for the CEC 2005 special session on real-parameter optimization*, 2005.

Index