

Métaheuristiques

Céline BUGAUD

Marco TESSARI

Jérôme POUILLER

Octobre 2004

Table des matières

1	Introduction	2
2	Problème discret : les phrases réflexives	3
2.1	Description du problème	3
2.2	Rappels sur les algorithmes génétiques	3
2.2.1	Diversification	4
2.2.2	Sélection	4
2.2.3	Récapitulatif des paramètres	4
2.3	Implémentation	4
2.3.1	Initialisation de la population	4
2.3.2	Choix des paramètres	5
2.3.3	Fonction d'erreur	5
2.3.4	Opérations sur le génomes	5
2.4	Résultats obtenus	5
2.4.1	Résultats des tests	5
2.4.2	Conclusions	6
3	Problème continu : étude de fonctions	7
3.1	Description du problème	7
3.2	Rappels sur le recuit simulé	8
3.2.1	Récapitulatif des paramètres	8
3.3	Implémentation	8
3.4	Résultats obtenus	9
4	Conclusion	12
	Bibliographie	13

Chapitre 1

Introduction

Ce rapport est voué à l'étude de deux problèmes d'optimisation. Un problème discret que nous avons résolu à l'aide d'un algorithme génétique, et un problème continu résolu à l'aide d'un recuit simulé.

Le problème discret proposé dans le cadre du TP était celui du placement de composants sur une puce. Nous avons choisi d'étudier un autre problème, celui des phrases réflexives. Une phrase réflexive est une phrase se décrivant elle-même. Par exemple : *Cette phrase contient 1 "0", 7 "1", 3 "2", 2 "3", 1 "4", 1 "5", 1 "6", 2 "7", 1 "8" et 1 "9"*. Un algorithme génétique a pour rôle de trouver les coefficients adéquats pour chaque chiffre.

Pour le problème continu, nous avons gardé l'étude des fonctions données en TP. Un recuit simulé se charge de trouver le minimum de ces fonctions. Nous allons donc présenter les résultats pour les neuf fonctions données, mais le recuit est sensé fonctionner avec les mêmes paramètres pour toute autre fonction...

Chapitre 2

Problème discret : les phrases réflexives

2.1 Description du problème

Le problème que nous allons essayer de résoudre est celui des phrases réflexives. Une phrase réflexive est définie comme une phrase annonçant le nombre de lettres ou de chiffres qu'elle contient.

Il y a 6 poules et 1 coq dans le poulailler. Dans cette phrase il y a : 1 0, 7 1, 4 2, 1 3, 2 4, 1 5, 2 6, 2 7, 1 8, 1 9.

La phrase ci-dessus est une phrase réflexive comptant les chiffres. La phrase est composée de deux parties, la première partie introduit le contexte et peut être considérée comme une initialisation. C'est la partie statique de la phrase. Dans notre cas c'est : *Il y a 6 poules et 1 coq dans le poulailler*. La deuxième partie annonce le décompte des chiffres.

Un *coefficient* désigne le nombre de fois qu'apparaît un chiffre, par exemple dans la phrase d'exemple 7 a pour coefficient 2.

Le but de notre programme est de trouver une solution au problème pour une initialisation donnée.

2.2 Rappels sur les algorithmes génétiques

Les algorithmes génétiques se sont inspirés du fonctionnement de l'évolution des espèces.

L'algorithme est basé sur la manipulation d'une population, c'est à dire d'un ensemble d'individus. Chaque individu est caractérisé par un génome et représente une solution au problème.

Le génome est un vecteur de gènes. La taille de ce vecteur et la forme des gènes sont dépendantes du problème. Par exemple, pour le problème du voyageur de commerce, chaque gène représenterait une ville à visiter, et la taille du génome serait le nombre de villes à visiter.

On doit aussi avoir une fonction appelée *fitness*, qui est fonction du génome, et que l'algorithme cherchera à maximiser. La plupart des problèmes sont à la base des problèmes de minimisation de fonctions (fonctions d'erreur très souvent), mais il est très simple de transformer la fonction à minimiser pour en faire une fonction fitness ($\frac{1}{f(x)}$, par exemple).

A l'initialisation de l'algorithme, on crée N individus pour former la population. Ces individus peuvent être générés au hasard ou selon certaines heuristiques, selon le problème, afin de faciliter la convergence.

Ensuite, chaque génération (itération de l'algorithme) comporte deux phases : la diversification et la sélection.

2.2.1 Diversification

Durant cette phase, l'algorithme crée de nouveaux individus par *cross-over* ou par *mutation*. La taille de la population augmente de M, le nombre d'individus créés.

Le *cross-over* (croisement) prend deux individus et les mélange pour en fabriquer deux nouveaux. Typiquement, il utilise un ou deux pivots sur les génomes des parents, et fabrique les enfants en alternant les génomes des parents.

La *mutation* fabrique un enfant à partir d'un parent. Elle est indispensable à la convergence de l'algorithme, sans elle la plupart du temps, on ne peut pas couvrir tout le domaine des solutions.

En général, elle se contente de copier chez l'enfant le génome du parent en en altérant un gène.

La mutation est appliquée en général avec une probabilité moindre que celle du *cross-over*.

2.2.2 Sélection

Après la phase de diversification, la sélection a pour but de réduire la taille de la population pour qu'elle redevienne la même qu'au début de l'algorithme. Pour ça on choisit un à un N individus parmi les N+M individus disponibles, sans remise et avec une probabilité fonction du *fitness* de l'individu : plus l'individu était bon, meilleures sont ses chances d'être conservé.

Deux mécanismes spéciaux sont couramment utilisés durant la phase de sélection : l'élitisme oblige le meilleur individu de la population à être sélectionné pour passer à la génération suivante, et la diversification empêche que trop d'individus soient identiques au sein d'une même population.

2.2.3 Récapitulatif des paramètres

Pour chaque problème que l'on voudra résoudre à l'aide d'un algorithme génétique, on aura donc besoin de donner les paramètres suivants :

- Taille du génome, type des gènes.
- Fonction de fitness. (ou d'erreur, et la transformer)
- Nombre maximal de génération.
- N : Taille initiale de la population.
- M : Nombre d'individus générés à chaque génération.
- Probabilité d'effectuer une mutation plutôt qu'un *cross-over*.
- Utilisation de l'élitisme (vrai ou faux).
- Utilisation de la diversification (vrai ou faux).
- On pourra aussi vouloir redéfinir les opérateurs de mutation et de *cross-over*.

2.3 Implémentation

Notre algorithme génétique est codé en java. Un individu (Biomorph en anglais) est représenté par une classe abstraite. Pour coder notre problème particulier, nous avons donc eu à dériver cette classe.

2.3.1 Initialisation de la population

Les premiers individus créés par l'algorithme ont un génome généré au hasard. La seule règle à suivre est que la valeur d'un gène est supérieure ou égale à la valeur de l'initialisation, afin de ne pas sortir du domaine de définition.

2.3.2 Choix des paramètres

Pour paramétrer l'algorithme, nous avons fait les choix suivants :

- Le génome représente une solution : dans notre cas c'est donc un vecteur d'entier. Sa taille est de dix (nombre de chiffres en décimal).
- La fonction de fitness est fabriquée 'artificiellement' à partir de la fonction d'erreur (cf plus bas).
- La taille de la population et le nombre d'individus générés à chaque itération sont fixés par l'utilisateur au lancement de l'algorithme, ainsi que la probabilité des mutations. Cela permet de faire plus facilement des tests pour ces paramètres.
- Le nombre maximal de générations peut être fixé par l'utilisateur. Sinon il peut choisir de faire tourner l'algorithme jusqu'à ce qu'il trouve une solution optimale.

2.3.3 Fonction d'erreur

La fonction d'erreur que nous avons choisi est la somme des carrés des erreurs de chaque gène :

$$erreur(\text{génome}) = \sum_{i=1}^{\text{génome.taille}} (\text{génome}[i] - \text{réelles}[i])^2$$

Avec réelle un vecteur de même taille que le génome et qui contient le nombre réel de chaque chiffre (ou lettre) de la phrase représentée par l'individu.

Cette fonction d'erreur est bien sûr à minimiser. Notre fonction fitness vaut donc :

$$fitness(\text{génome}) = \frac{1}{erreur(\text{génome}) + 1}$$

Le '+1' étant là uniquement pour éviter une division par zéro quand la solution est optimale ($erreur = 0$).

2.3.4 Opérations sur le génomes

Nous n'avons pas touché sur ce problème à l'opérateur de cross-over. Il effectue l'opération basique en utilisant un pivot.

Pour la mutation, nous altérons un gène au hasard en lui ajoutant ou retirant un nombre aléatoire entre 1 et une constante. (Cette constante peut valoir 1...). Cette opération est effectuée évidemment dans la limite du domaine du problème (par exemple, pas de nombres négatifs).

2.4 Résultats obtenus

2.4.1 Résultats des tests

Dans cette partie, nous nous proposons de montrer les résultats obtenus sur ce problème, en fonction des différents paramètres donnés à l'algorithme.

Les tests que nous avons effectués ne comporte pas de phrase d'initialisation, mais les résultats avec ou sans initialisation sont sensiblement identiques.

Nous avons pour chaque test lancé vingt fois l'algorithme avec les mêmes paramètres. Les valeurs de nos résultats sont la moyenne des résultats des vingt exécutions.

Le tableau suivant montre pour un certains nombre de combinaisons de parametres les résultats de l'algorithme : le nombre de générations moyen qu'il a fallu pour trouver le résultat et le nombre moyen d'individus qui ont été créés pour y parvenir.

N	M	E	D	P(m)	G moy	G max	G min	I
200	200	oui	oui	0.2	94.35	156	22	18870
200	200	non	oui	0.2	89.1	207	33	17820
200	200	oui	oui	0.4	74.6	131	21	14920
200	200	oui	oui	0.6	56.1	97	20	11220
200	200	non	oui	0.6	59.05	107	31	11810
50	200	oui	oui	0.2	166.7	424	31	33340
200	50	oui	oui	0.2	388.4	649	210	19420
50	50	oui	oui	0.2	444.15	1177	154	22207.5
50	200	oui	oui	0.6	64.9	173	11	12980
200	50	oui	oui	0.6	197.1	358	71	9855
50	50	oui	oui	0.6	169.65	370	53	8482.5
100	100	oui	oui	1	64.25	102	24	6425
100	100	non	oui	1	63.55	124	31	6355

avec :

N : Taille de la population initiale.

M : Nombre d'individus créés par génération.

E : Utilisation du mécanisme d'élitisme.

D : Utilisation du mécanisme de diversification.

P(m) : Probabilité de mutation.

G moy : Nombre moyen de générations effectuées pour arriver au résultat.

G max : Nombre maximal de générations effectuées pour arriver au résultat.

G min : Nombre minimal de générations effectuées pour arriver au résultat.

I : Nombre moyen d'individus créés pour arriver au résultat.

2.4.2 Conclusions

On remarque que tous les tests utilisent le mécanisme de diversification. En fait, aucun des tests que nous avons lancé sans ce mécanisme n'a réussi à trouver de bonne solution, ils stagnaient tous avec une erreur de 1. On voit donc l'importance de ce mécanisme. Quand il n'est pas utilisé, tous les individus ont tendance à se regrouper dans un ou plusieurs minima locaux et ont beaucoup de mal à s'en sortir (s'ils s'en sortent).

Des autres résultats listés ci-dessus, on remarque plusieurs choses : déjà que l'algorithme converge plus vite avec une probabilité de mutation plus forte. Ensuite, on voit que la taille de la population initiale (et donc la taille de la population au cours de l'algorithme) ne doit pas être trop petite, l'algorithme converge moins bien avec une population initiale de 50 individus qu'avec une population initiale de 200 individus. On voit aussi que le mécanisme d'élitisme a tendance à améliorer légèrement la convergence.

Au final, l'analyse des deux derniers tests montre qu'un algorithme génétique n'était peut être pas la meilleure solution pour régler ce problème : le fait de mettre la probabilité de mutation à 1 a considérablement amélioré la convergence. Or cela signifie perdre le principe de la génétique : on n'a plus que des mutations et aucuns croisements. On est alors dans une sorte de recuit simulé avec plusieurs solutions examinées en parallèle. On aurait donc peut être eu intérêt à tenter de résoudre ce problème à l'aide d'un recuit simulé et non d'un algorithme génétique...

Chapitre 3

Problème continu : étude de fonctions

3.1 Description du problème

Le problème est ici un problème d'optimisation de fonctions. Nous avons essayé d'implémenter un algorithme de recuit simulé qui soit capable d'approximer le minimum global d'à peu près n'importe quelle fonction.

Ce programme a été testé et implémenté pour le moment pour huit fonctions connues, qui regroupent à peu près toutes les difficultés que l'on peut rencontrer dans la recherche de minima de fonctions.

– Michalewicz :

$$MZ(x) = \sum_{i=1}^n \sin(x_i) * \left[\sin\left(\frac{i * x_i^2}{\pi}\right) \right]^{20}$$

– De Jong :

$$F1(x) = \sum_{i=1}^n x_i^2$$

– De Jong 2 :

$$F2(x) = 100 * (x^2 - y)^2 + (1 - x)^2$$

– De Jong 3 :

$$F3(x) = \sum_{i=1}^n \text{floor}(x_i)$$

– Goldstein and Price :

$$GP(x) = [1 + (x + y + 1)^2 * (19 - 14x + 3x^2 - 14y + 6xy + 3y^2)] * \\ [30 + (2x - 3y)^2 * (18 - 32x + 12x^2 + 48y - 36xy + 27y^2)]$$

– Rosenbrock :

$$\sum_{i=1}^{n-1} 100 * (x_i^2 - x_{i+1})^2 + (1 - x_i)^2$$

– Zakharov :

$$Z(x) = \sum_{i=1}^n x_i^2 + \left[\sum_{i=1}^n 0.5 * i * x_i \right]^2 + \left[\sum_{i=1}^n 0.5 * i * x_i \right]^4$$

– Schwefel :

$$\sum_{i=1}^n -x_i * \sin(\sqrt{|x_i|})$$

Les résultats obtenus pour chacune de ces fonctions sont décrits plus bas.

3.2 Rappels sur le recuit simulé

Le recuit simulé s'est basé sur des principes de la métallurgie et de la recuite des métaux. Le but est de faire baisser petit à petit une *température* pour que l'énergie se stabilise dans un état minimal.

Appliqué à l'informatique, ça revient à prendre une solution à un problème et l'altérer. A chaque altération, on accepte la solution si son énergie est plus faible, sinon on l'accepte avec la probabilité $e^{-\frac{\delta}{\text{température}}}$, c'est la règle d'acceptation de métropolis.

On voit donc que plus la température est haute, plus une solution éronée a des chances d'être choisie, ce qui permet de sortir des minima locaux. Si cette règle n'était pas mise en place, le résultat de l'algorithme serait toujours le minimal local qui se trouve dans la même vallée que la solution de départ.

Au fur et à mesure des itérations de l'algorithme, la température est abaissée progressivement, les perturbations dégradantes sont donc de moins en moins acceptées.

Les conditions d'arrêt de l'algorithme sont les suivantes :

- Tous les $100 * N$ tours de boucles (avec N le nombre de variables du problème), on abaisse la température.
- Quand durant au moins 3 paliers de température, aucune nouvelle solution n'a été acceptée, l'algorithme s'arrête.

3.2.1 Récapitulatif des paramètres

Pour chaque problème que l'on voudra résoudre à l'aide d'un recuit simulé, on aura donc besoin de donner les paramètres suivants :

- Type des solutions et nombre de variables.
- Fonction d'énergie.
- Opérateur de perturbation de la solution.
- Température initiale.

3.3 Implémentation

Comme l'algorithme génétique, notre recuit simulé est codé en java. Une class Solution représente une solution au problème et définit toutes les données du problème : elle s'occupe de générer des nouvelles solution en perturbant la solution courante, et donne à l'algorithme toutes les informations spécifiques au problème dont il a besoin.

Ici, notre classe de solution utilise une fonction qui lui est passée en parametre et qui permet de connaître :

- Le nombre de variables.
- Le domaine de définition des variables.
- La valeur de la fonction étant donné la valeur de ses variables.

Nous avons rajouté une nouvelle condition d'arrêt à l'algorithme, si la solution change mais pas l'énergie au cours de plusieurs paliers de température. Ce changement nous a été dicté par une des fonctions données plus haut dont la forme est en paliers : une infinité de solutions différentes donnent un résultat optimal, or quand l'énergie est inchangée, le résultat est habituellement accepté, donc l'algorithme ne pouvaient pas terminer.

Les solutions initiales données à l'algorithme sont prises au hasard dans le domaine de définition des variables.

Pour perturber nos solutions, nous suivons la méthode suivante : un tiers des variables est affecté par la perturbation. A chacune des variables est affecté un domaine de variation appelé

STEP. Les variables sont perturbées comme suit :

$$x_i = x_i + z * STEP_i$$

avec $z \in [-1, 1]$.

A chaque paliers de température, ces domaines de perturbation sont réajustés en fonction du nombre de perturbations acceptées par variables. Si une variable a fait partie de beaucoup de perturbations acceptées, son domaine de perturbation est doublé. Si au contraire, les perturbations dont elle a fait partie ont été très peu acceptées, son domaine de perturbation est diminué de moitié. Ce système permet d'avancer plus vite dans le domaine de définition de la variable quand elle est loin du minimum, puis d'affiner la recherche quand elle s'en est rapprochée.

3.4 Résultats obtenus

Dans cette section nous allons présenter les résultats des tests effectués sur ce problème. Les tests portent sur les fonctions décrites plus haut. Pour chaque test, l'algorithme a été relancé vingt fois. Le résultat présenté est celui de la meilleure solution trouvée. Les résultats sont tronqués à la deuxième décimale.

Les tests que nous présentons ici ont tous été réalisés avec deux variables. Nous avons réalisés quelques tests avec plus de variables mais ces tests prennent beaucoup plus de temps, nous avons donc préféré avoir plus de tests avec moins de variables. Pour les quelques tests que nous avons effectué avec 10 variables sur chaque fonction, l'algorithme a trouvé une réponse proche du résultat obtenu pour 2 variables.

f	T	N	E	x	y	I
Michalewicz	5000	2	0	1.39	0	655950
Michalewicz	1000	2	0	2.27	0	671210
Michalewicz	200	2	0	1.49	0	613570
Michalewicz	100	2	0	2.41	0	630090
Michalewicz	50	2	0	0.72	0	680930
De Jong 1	5000	2	0	0	0	47430
De Jong 1	1000	2	0	0	0	44430
De Jong 1	200	2	0	0	0	41170
De Jong 1	100	2	0	0	0	39110
De Jong 1	50	2	0	0	0	37630
De Jong 2	5000	-	0	1	1	42020
De Jong 2	1000	-	0	1	1	19260
De Jong 2	200	-	0	1	1.01	35040
De Jong 2	100	-	0	0.99	0.99	33720
De Jong 2	50	-	0	1	0.99	32570
De Jong 3	5000	2	-12	-5.02	-5.12	22580
De Jong 3	1000	2	-12	-5.08	-5.10	19260
De Jong 3	200	2	-12	-5	-5.07	16280
De Jong 3	100	2	-12	-5.09	-5.04	14690
De Jong 3	50	2	-12	-5.07	-5	13410
Goldstein & Price	5000	-	3	0	-1	39210
Goldstein & Price	1000	-	3	0	-1	35660
Goldstein & Price	200	-	3	0	-1	34860
Goldstein & Price	100	-	3	0	-1	30700
Goldstein & Price	50	-	3	0	-1	28980
Rosenbrock	5000	2	0	1	1	40820
Rosenbrock	1000	2	0	0.99	0.98	39510
Rosenbrock	200	2	0	0.99	0.99	34860
Rosenbrock	100	2	0	1.02	1.05	34270
Rosenbrock	50	2	0	1.01	1.02	34000
Zakharov	5000	2	0	0	0	47690
Zakharov	1000	2	0	0	0	29210
Zakharov	200	2	0	0	0	40270
Zakharov	100	2	0	0	0	39560
Zakharov	50	2	0	0	0	37810
Schwefel	5000	2	-837.97	420.92	420.97	32070
Schwefel	5000	2	-837.97	420.98	420.95	29210
Schwefel	200	2	-837.97	421.01	420.97	26850
Schwefel	5000	2	-837.97	420.91	420.98	24720
Schwefel	5000	2	-837.97	420.99	420.94	22660

Avec :

f : Nom de la fonction.

T : Température initiale.

N : Nombre de variables.

E : Meilleure énergie trouvée.

x : Coordonée x de la meilleure énergie.

y : Coordonée y de la meilleure énergie.

I : Nombre moyen d'itérations (=perturbations) pour arriver au résultat.

La table suivante rappelle les minimums globaux 'réels' de ces fonctions.

f	N	E	x	y
Michalewicz	2	-1.80	2.19	1.57
De Jong 1	2	0	0	0
De Jong 2	-	0	1	1
De Jong 3	2	-12	-5.12	-5.12
Goldstein & Price	-	3	0	-1
Rosenbrock	2	0	1	1
Zakharov	2	0	0	0
Schwefel	2	-837.96	420.96	420.96

On voit qu'à part pour la première fonction - Michalewicz, l'algorithme trouve des résultats très proches des minimas locaux réels. En fait, nous pensons que nous avons une erreur dans la formule de cette fonction, car elle ne semble pas vouloir produire de nombres négatifs, et les valeurs de x et y du minimal global donné ci dessus donne 0 comme résultat pour notre fonction.

Les résultats de l'algorithme ne changent pas de manière visible selon les différentes valeurs de température initiale que nous avons testé. On voit simplement que le nombre d'itérations moyen baisse en même temps que la température initiale.

Chapitre 4

Conclusion

Nous avons présenté ici deux exemples d'applications des métaheuristiques : l'utilisation d'un algorithme génétique pour un problème discret, celui des phrases réflexives et l'utilisation d'un recuit simulé pour un problème continu, la recherche de minimal global de fonctions.

Au final les deux applications ont fonctionné sur les tests que nous avons effectué, même si nous avons vu que l'algorithme génétique n'était peut être pas le plus indiqué sur le problème des phrases réflexives.

Bibliographie

- [1] D. R. HOFSTADER, *Ma thémagie*, InterEditions, Paris, France, 1988. Traduction française de [2] par Jean-Baptiste Bertelin.
- [2] D. R. HOFSTADTER, *Metamagical Themas : Questing for the Essence of Mind and Patterns*, Bantam Books, New York, USA, 1986. Présentation du livre disponible sur http://en.wikipedia.org/wiki/Metamagical_Themas.
- [3] J.-B. ROUX, *Cinq c cinq i cinq n cinq q*. Disponible sur <http://hypo.ge-dip.etat-ge.ch/www/math/html/node92.html>. Juste un mot sur les phrases reflexives.