Methods and Heuristics for Learning and Optimization
Exercise 6: Genetic Algorithms and Function Minimization

Romain Mencattini

6 décembre 2016

Table des matières

1 Formalisation du problème

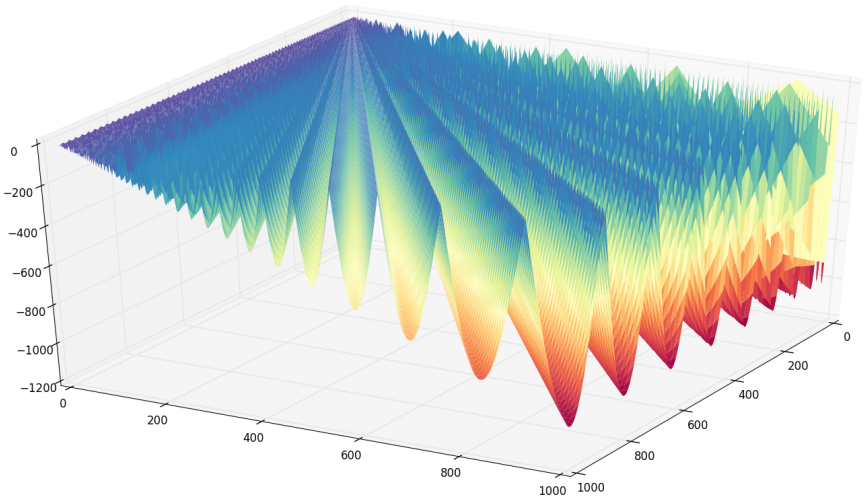
Dans ce travail, il nous est demandé de minimiser une fonction continue :

$$f(x,y) = -\left|\frac{1}{2}x\ sin(\sqrt{|x|})\right| - \left|y\ sin(30\sqrt{\left|\frac{x}{y}\right|})\right|$$

Avant de commencer l'optimisation, il est intéressant de visualiser la fonction afin de voir à quel paysage nous allons avoir affaire. Si on prend comme borne :

$$x, y\ \in\ [10, 1000]\ \mathbb{N}$$

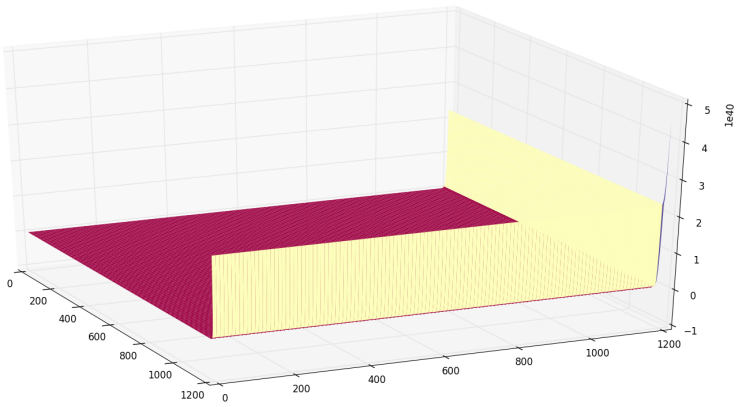
Voici donc ce que donne cette fonction avec des entiers compris entre les bornes susnommées :



On remarque qu'il s'agit d'un paysage extrêmement rugueux. Avec beaucoup de variations ainsi que beaucoup de minimas locaux. Nous n'allons cependant pas pouvoir utiliser la fitness en l'état. En effet, si l'on étend le domaine, il y a des endroits avec une fitness encore plus basses. La métaheuristique risque donc d'être attirée par ces zones. Il faut pénaliser les dites zones afin d'éviter ces problèmes. Pour ce faire nous allons utiliser une fitness légèrement modifiée :

$$f(x,y) = (\tfrac{x}{1000})^{512} + (\tfrac{10}{x})^4 + (\tfrac{y}{1000})^{512} + (\tfrac{10}{y})^4 - \left|\frac{1}{2}x\ sin(\sqrt{|x|})\right| - \left|y\ sin(30\sqrt{\left|\frac{x}{y}\right|})\right|$$

Lors que x et y se trouvent $\in [10, 1000]$, alors les fractions sont ≤ 1 ce qui signifie que les puissances seront bornées par 1. Cependant si les nombres sortent des bornes, alors les valeurs vont explosées ainsi que la fitness ; pénalisant de ce fait, les nombres hors des limites. Nous n'avons pas géré les nombres négatifs, car nous travaillons dans $\mathbb{N}$. Nous avons utilisé des puissances de 2 afin de gagner quelques nano secondes. En effet, les puissances de 2 peuvent se calculer grâce à l'exponentiation rapide qui est légèrement plus rapide que les autres exponentiation. On obtient donc ce graph :



2 Algorithme Génétique

Nous allons présenté le pseudo-code puis nous attarder sur chaque partie :

1. Génération d'une population
2. Sélection
3. CrossOver
4. Mutation
5. Retour à 2. tant que la condition n'a pas été atteinte

Pour générer une population, nous avons besoin de la taille, soit n le nombre d'individus. La probabilité de mutation, soit p_m. La probabilité de crossover, soit p_c. Nous verrons les valeurs qui peuvent leur être attribuées dans la section des résultats. Nous générons donc des individus avec un x et un y, pour une population. Ces individus auront une certaine probabilité de muter ou de croiser. Il y a une petite subtilité dans le codage de x, y. Ils sont représentés par une chaîne de bits. Afin de faciliter la mutation et le crossver.

L'opération de sélection consiste à choisir n individus afin de construire une nouvelle population. On peut formaliser cela par :

$$P' = Selection(P)$$

Il y a plusieurs manières de procéder. Nous allons appliquer un tournois. Pour ce faire, nous allons effectuer n fois ces opérations :

— choisir k individus dans P avec remise et de manière uniforme

— prendre celui avec la meilleure fitness

Dans le cadre de ce travail, nous fixons $k = 5$.

Le CrossOver s'effectue sur la population obtenue à partir de la sélection :

$$P'' = CrossOver(P')$$

Pour effectuer un crossover, nous allons prendre les individus de la population deux à deux. Puis, une fois qu'on a $individu_i$, $individu_{i+1}$. On prend chacune des composantes x,y. C'est sur ces éléments que nous allons appliquer la transformation. Soit x_1,x_2, nous allons couper au milieu de leur représentation binaire et échanger les parties :

$$x_1 = 000000$$
$$x_2 = 111111$$
$$x_1\text{'} = 000111$$
$$x_2\text{'} = 111000$$

On fait la même chose pour y_1,y_2, et on obtient nos nouveaux individus. On applique cela sur toutes la population P'.

L'opération de mutation consiste à changer ,avec une certaine probabilité p_m, les bits de x,y. On va donc parcourir chaque bits des composantes des individus de la population, on tire un nombre aléatoire $\in [0, 1[$, si le nombre est plus petit, on change le bit sinon on ne fait rien. En faisant cela on obtient une nouvelle population :

$$P''' = Mutaiton(P'')$$

3 Résultats et Conclusion

C'est une constante dans les métaheuristiques : il y a de nombreux paramètres et donc de nombreuses manières de faire varier l'algorithme. Dans le cadre de ce travail, nous en avons testé certaines.

Soit le nombre d'évaluation de la fitness n_f, donné par $|population| * loop$, où $loop$ est le nombre de boucle. Les résultats sont donnés en terme de pourcentage d'obtention de l'optimum global qui se trouve en $x = 903$,$y = 917$, (trouvé par recherche exhaustive). Les données obtenues sont une moyenne sur 1000 runs.

Pour $n_f = 10^3$, avec $|population| = 100$ et $loop = 10$:

	avec CrossOver	sans CrossOver
pm = 0.1	1.2	1.8
pm = 0.01	2	0.8

Pour $n_f = 10^4$, avec $|population| = 100$ et $loop = 100$:

	avec CrossOver	sans CrossOver
pm = 0.1	7.39	9.8
pm = 0.01	7.6	7.6

Pour $n_f = 10^5$, avec $|population| = 100$ et $loop = 1000$:

	avec CrossOver	sans CrossOver
pm = 0.1	10.4	11.2
pm = 0.01	11.4	8.6

Les remarques que l'on peut faire sont :

- l'augmentation de n_f permet d'augmenter la probabilité d'obtenir le minimum global.
- les résultats ne semble pas très bon, car on ne trouve qu'avec une probabilité $\in [0.8, 11.4]$
- on obtient les meilleures résultats soit avec :
  - $p_m = 0.1$ et sans CrossOver
  - $p_m = 0.01$ et avec CrossOver

  Nous pensons que cela montre le fait, que si la mutation est trop violente ($p_m = 0.1$), alors il ne faut pas rajouter d'aléatoire dans l'algorithme. Cependant si p_m est plus faible, alors on peut ajouter un crossover.

Afin de pousser un peu les expérimentations, nous voulions voire si nous étions loin de l'optimal lorsque nous trouvions une solution. En effet, il peut être intéressant de dire : si nous trouvons une solution qui est proche à 1% près de la solution optimale, c'est une solution acceptable. Voyons donc pour $p_m = \{0.1, 0.01\}$, $n_f = \{10^3, 10^4, 10^5\}$, $loop = \{10, 100, 1000\}$ et $|population| = 100$. On obtient 100% dans tout les cas. Cela peut paraître étonnant, mais on est toujours à moins d'un poucent de la solution optimale. L'algorithme peut sembler intéressant dans ce sens.

Nous avons voulu pousser un peu plus loin, en n'acceptant que les solutions proches de 0.1%, pour les mêmes paramètres que cités plus haut. Nous avons obtenu entre 90 et 95% une solution proche de 0.1% de l'optimum. Ce qui est une bonne nouvelle.

L'algorithme a donc de bons résultats lorsqu'il s'agit de trouver un résultat proche de l'optimal. Sans compter le fait que le paysage de la fonction est extrêmement rugueux, donc c'est une preuve de résistance au paysage.

Cependant, nous étions déçu par les résultats brutes, nous avons donc voulu changer la taille de la population pour voir l'influence que celle-ci peut avoir. Nous avons donc refait les mesures pour $p_m = \{0.1, 0.01\}$, $n_f = \{10^3, 10^4, 10^5\}$, $loop = \{5, 50, 500\}$ et $|population| = 200$. On garde donc le même nombre d'évaluation de fitness, afin d'avoir un point de comparaison avec une population de taille 100. Nous avons obtenu :

	$n_f = 10^3$	$n_f = 10^4$	$n_f = 10^5$
pm = 0.1	1	17	20.4
pm = 0.01	3	14.6	14.8

On observe que les résultats sont arbitrairement meilleurs que lors de la population à 100 individus. En plus, on garde le même effort computationnel ce qui fait que le temps reste le même. Il peut donc être intéressant de faire varier la taille de la population plus que de faire varier le nombre d'exécution. En exécutant, la mesure à 0.1% de distance, on obtient des résultats de 95% à 98%, ce qui est également une bonne chose.