Université Hassan II de Casablanca

Faculté des sciences et techniques de Mohammedia

Sujet : optimisation contrainte avec des algorithmes génétiques

Fait par:
Reda Lamtoueh

LST IRM

Année Universitaire :2020-2021

Contents

Abstrait

Les algorithmes génétiques sont des algorithmes évolutionnaires bien adaptés à la recherche de solutions globales à la nature variée des problèmes d'optimisation. Les inspirations dans le développement de l'AG sont dérivées du principe de fonctionnement de la génétique naturelle. Les opérateurs tels que la reproduction, le croisement et la mutation sont employés de la même manière que la génétique naturelle. Ces étapes impliquaient des éléments de probabilité qui rendaient la recherche de la solution optimale aléatoire, rendant l'AG stochastique et non déterministe. Plusieurs initiatives ont été prises par les chercheurs pour améliorer la direction de la recherche et la rendre plus définitive. Le présent travail vise à proposer une nouvelle approche par étapes dans la sélection des intervalles de recherche de variables à l'aide d'un algorithme génétique.

1 Introduction

Les problèmes d'optimisation sont caractérisés par des fonctions objectives avec ou sans contraintes. Dans l'optimisation contrainte, il existe des possibilités de combinaisons de fonctions objectives linéaires et non linéaires avec des contraintes linéaires et non linéaires. Il existe différentes méthodes rapportées, qui répondent à ces problèmes d'optimisation. Chaque méthode a ses limites et peut être appliquée à certaines situations de manière sélective. En plus de cela, ces techniques se révèlent inefficaces et arrivent souvent à un optimum relatif qui est le plus proche du point de départ. L'algorithme génétique est unique parmi les méthodes d'optimisation et est devenu un outil universel qui peut être appliqué aux divers problèmes d'optimisation. Où la fonction objectif et les contraintes sont bien définies et GA a atteint l'optimum global avec une probabilité élevée .

2 Les algorithmes génétiques

Les algorithmes génétiques (AG) relèvent de la catégorie des algorithmes évolutionnaires dont le principe de fonctionnement est basé sur la mécanique de la génétique naturelle. L'objectif fondamental de la génétique naturelle est la rétention des gènes adaptés et l'élimination des gènes redondants. Nouvelles générations créées en manipulant le code génétique à l'aide d'outils tels que la sélection, le croisement et la mutation. GA fonctionne également de manière similaire dans le but de rechercher une solution appropriée aux problèmes impliquant soit la minimisation, soit la maximisation de la fonction objectif. GA utilise des outils similaires tels que la sélection, le croisement et la mutation appliqués à une population de chaînes binaires générées aléatoirement. À chaque génération, un nouvel ensemble d'espèces ou de cordes artificielles est créé en utilisant des morceaux des plus aptes parmi les anciens; une nouvelle pièce occasionnelle peut être essayée pour faire bonne mesure. Il a été prouvé que les algorithmes génétiques permettent une recherche robuste dans des espaces complexes. L'algorithme génétique diffère des autres procédures d'optimisation et de recherche des manières suivantes:

- La recherche est effectuée sur une population générée aléatoirement pour une combinaison de variables d'une solution possible en manipulant leur version codée en binaire
- GA peut être vue comme une technique universelle qui peut répondre à plusieurs types de problèmes d'optimisation et gérer des fonctions non linéaires, complexes et bruitées.
- GA effectue une recherche globale et arrive très souvent à ou près de l'optimum global.
- GA n'impose pas de conditions préalables à la fonction telles que la régularité, la dérivabilité et la continuité.

3 Principe des algorithmes Génétiques

3.1 Codage

L'algorithme génétique est inspiré du modèle de la génétique biologique et la plupart de sa terminologie a été empruntée à la génétique. Le génotype est l'ensemble des gènes possédés par un individu et les gènes sont formés d'ADN. Dans un organisme vivant, l'ADN se combine avec des protéines pour former les gènes. Les gènes ont une valeur appelée allèles. Chaque gène a une position unique sur le chromosome appelée locus. Les encodages sont choisis selon les propriétés suivantes :

- Incarne les blocs de construction fondamentaux qui sont importants pour le type de problème.
- Se prête aux opérateurs génétiques qui peuvent propager ces blocs de construction du génotype des parents au génotype de la progéniture.
- Permet une cartographie traitable au phénotype.

3.1.1 Codage Binaire

C'est la forme d'encodage la plus utilisable. Dans ce codage, chaque chromosome est représenté comme d'une chaîne binaire. Dans le codage binaire, chaque chromosome est une chaîne de bits, 0 ou 1.

Chromosome1	110101110010
Chromosome2	000101010011

Figure 1: Codage Binaire.

Dans ce type de codage, chaque bit représente une caractéristique du solution. De l'autre côté, chaque chaîne binaire représente une valeur. Avec un plus petit nombre d'allèles, un certain nombre de chromosomes peuvent être représentés. Les opérations de croisement possibles dans le codage binaire sont le croisement à 1 point, le croisement à Npoint, le croisement uniforme et le croisement arithmétique. L'opérateur de mutation possible est Flip. Dans la mutation Flip, les bits passent de 0 à 1 et de 1 à 0 en fonction du chromosome de mutation généré. Ceci est généralement utilisé dans le problème du sac à dos où le codage binaire est utilisé pour montrer la présence d'éléments, par exemple 1 pour indiquer la présence d'un élément et 0 pour l'absence d'un élément.

3.1.2 Codage par permutation

Dans de nombreux problèmes, la solution est représentée par un ordre d'éléments. Dans de tels cas, la représentation par permutation est la plus appropriée. Un exemple classique de cette représentation est le problème du voyageur de commerce (TSP). Dans ce cas, le vendeur doit faire le tour de toutes les villes, visiter chaque ville exactement une fois et revenir à la ville de départ. La distance totale du tour doit être minimisée. La solution à ce TSP est naturellement un ordre ou une permutation de toutes les villes et donc l'utilisation d'une représentation par permutation a du sens pour ce problème.

Chromosome1	0 1 2 3 4 5 6 7 8 9
Chromosome2	1 3 4 2 7 5 6 9 8 0

Figure 2: Codage par permutation.

3.1.3 Code Réel

Dans le codage réel, chaque chromosome est représenté comme une chaîne d'une certaine valeur. La valeur peut être un entier, un nombre réel, un caractère ou un objet. Dans le cas de valeurs entières, l'opérateur de croisement appliqué est le même que celui appliqué sur le codage binaire. Les valeurs peuvent être tout ce qui est lié au problème, aux nombres de formulaires, aux nombres réels ou aux caractères de certains objets compliqués.

Chromosome1	1.23, 2.12, 3.14, 0.34, 4.62
Chromosome2	'ABDJEIFJDHDDLDFLFEGT'

Figure 3: Codage Réel.

Le codage de valeur peut être utilisé dans les réseaux de neurones. Ce codage est généralement utilisé pour trouver des poids pour le réseau de neurones. La valeur du chromosome représente les poids correspondants pour les entrées.

3.2 Fonction de Fitness

La fonction de fitness est une fonction qui prend une solution candidate au problème en entrée et produit en sortie à quel point la solution est 'adaptée' ou 'bonne' par rapport au problème considéré. Le calcul de la valeur de fitness est effectué à plusieurs reprises dans un GA et doit donc être suffisamment rapide. Un calcul lent de la valeur de fitness peut affecter négativement un GA et le rendre exceptionnellement lent. Cependant, pour des problèmes plus complexes avec plusieurs objectifs et contraintes, un concepteur d'algorithmes peut choisir d'avoir une fonction de fitness différente. Une fonction de fitness doit posséder les caractéristiques suivantes:

- La fonction de fitness doit être suffisamment rapide pour être calculée.
- Il doit mesurer quantitativement l'adéquation d'une solution donnée ou la manière dont des individus adaptés peuvent être produits à partir de la solution donnée.

Dans certains cas, le calcul direct de la fonction de fitness peut ne pas être possible en raison de la complexité inhérente du problème à résoudre. Dans de tels cas, nous effectuons une approximation de la condition physique en fonction de nos besoins.

3.2.1 Fonction de Fitness pour les Problèmes sans contraints

Pour les problèmes d'optimisation sans contraints, la fonction de fitness et la fonction objectif sont les mêmes, car l'objectif est de maximiser ou de minimiser la fonction objectif donnée.

3.2.2 Fonction de Fitness pour les Problèmes sous contraints

Pour des problèmes plus complexes avec plusieurs objectifs et contraintes, un concepteur d'algorithmes peut choisir d'avoir une fonction de fitness différente de la fonction d'objectif. En général, un problème d'optimisation numérique contraint est défini comme suit:

$$\begin{aligned} \textit{optimiser} \quad & f(x) \qquad x \in R^n \\ \textit{sachant que} \quad & h_i(x) = 0 \quad i = 1, ...m \\ & g_i(x) < 0 \quad i = m+1, ...p \end{aligned}$$

Sans perte de généralité, nous pouvons transformer n'importe quel problème d'optimisation en un problème de minimisation et nous développons donc notre discussion en ces termes. Les contraintes définissent la région réalisable, ce qui signifie que si le vecteur x respecte toutes les contraintes $h_i(x) = 0$ et $g_i(x) < 0$

alors il appartient à la région réalisable. Les méthodes traditionnelles reposant sur le calcul différentiel exigent que les fonctions et les contraintes aient des caractéristiques bien particulières (continuité, dérivabilité, dérivabilité du second ordre, etc.) ; ceux basés sur les GA n'ont pas de telles limitations.

Dans les algorithmes génétiques, les contraintes sont principalement gérées en utilisant le concept de fonctions de pénalité, qui pénalisent les solutions irréalisables en réduisant leurs valeurs de fitness proportionnellement aux degrés de violation des contraintes. Dans la plupart des schémas de pénalités, certains coefficients ou constantes doivent être spécifiés au début du calcul. Étant donné que ces coefficients n'ont généralement pas de significations physiques claires, il est presque impossible d'estimer les valeurs appropriées de ces coefficients, même par expérience. De plus, la plupart des schémas utilisent des coefficients constants tout au long du calcul. Cela peut se traduire par une pénalité trop faible ou trop forte lors des différentes phases de l'évolution. par conséquent, la fonction Fitness est définie comme suit:

$$fitness(x) = \begin{cases} \text{f(x)} & x \in regionfaisable, \\ \text{f(x)+penalite(x)} & x \notin regionfaisable \end{cases}$$

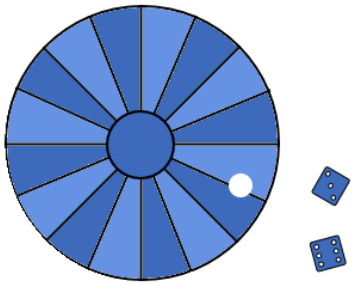
4 Les Opérateurs

4.1 La sélection

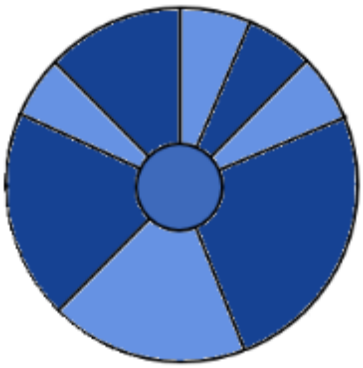
La sélection est l'étape d'un algorithme génétique dans laquelle des génomes individuels sont choisis dans une population pour une reproduction ultérieure (à l'aide de l'opérateur de croisement).

4.1.1 La roue de Loterie

La sélection par roulette est une méthode de sélection stochastique, où la probabilité de sélection d'un individu est proportionnelle à sa forme physique. La méthode est inspirée des roulettes du monde réel mais possède des distinctions importantes par rapport à celles-ci. Comme nous le savons dans les films sur les casinos et les jeux d'argent, les roulettes ont toujours des machines à sous de même taille:



Cela signifie cependant que tous les créneaux ont la même probabilité d'être sélectionnés. Au lieu de cela, nous pouvons implémenter une version pondérée de la roulette. Avec lui, plus la fitness d'un individu est grande, plus sa sélection est probable :



Le premier élément de la méthode de sélection à la roulette est donc que l'aptitude individuelle est proportionnelle à sa probabilité de sélection. Ce n'est pas suffisant cependant. En effet, si la population compte n individus, alors la somme des probabilités $\sum_{i=1}^{n} p_i$ de leurs sélections est égale à un. En conséquence, nous devons également normaliser toutes les valeurs des probabilités individuelles à l'intervalle [0,1]. Enfin, nous pouvons résumer les considérations faites ci-dessus et développer une méthode qui satisfait les exigences que nous nous sommes fixées.

Dans une population de n individus, pour chaque chromosome x avec une valeur de fitness correspondante f_x, nous calculons la probabilité correspondante f_x de sélection comme:

$$p_x = \frac{f_x}{\sum_{i=1}^{n} f_i}$$

4.1.2 Sélection par Rang

La sélection de rang fonctionne également avec des valeurs de fitness négatives et est principalement utilisée lorsque les individus de la population ont des valeurs de fitness très proches (cela se produit généralement à la fin de la course). Cela conduit à ce que chaque individu ait une part presque égale du gâteau (comme dans le cas d'une sélection proportionnelle à la forme physique) comme le montre l'image suivante et donc chaque individu, quelle que soit sa forme l'un par rapport à l'autre, a approximativement la même probabilité d'être sélectionné en tant que parent. Cela conduit à son tour à une perte de pression de sélection vers des individus plus en forme, ce qui oblige l'AG à faire de mauvaises sélections de parents dans de telles situations.

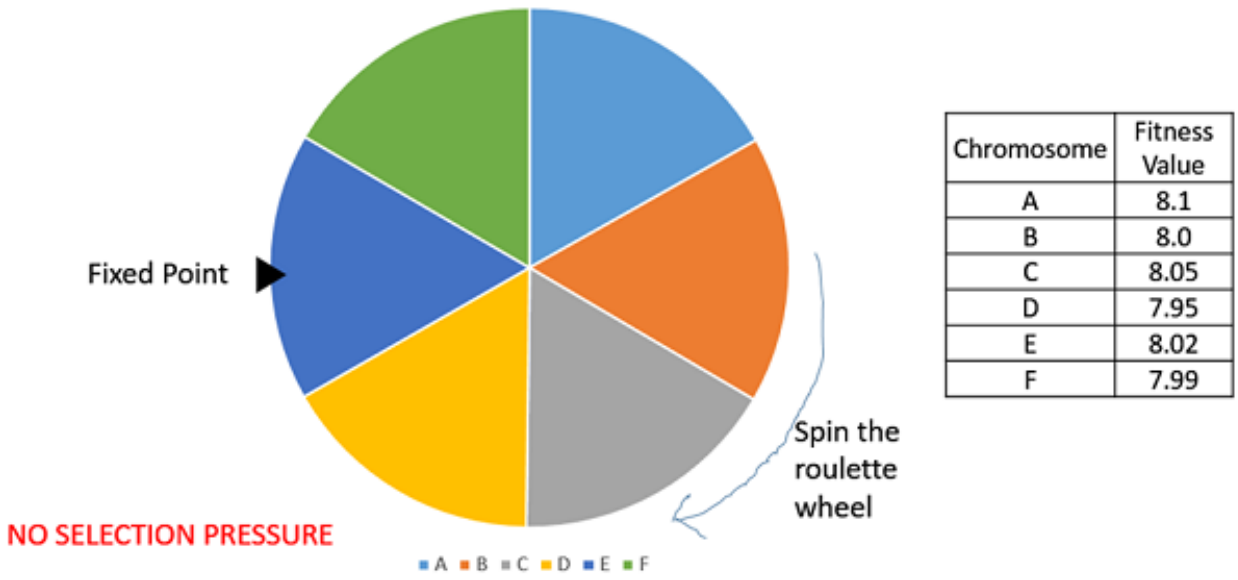


Figure 4: Sélection par Rang.

En cela, nous supprimons le concept de valeur de fitness lors de la sélection d'un parent. Cependant, chaque individu de la population est classé en fonction de sa condition physique. La sélection des parents dépend du rang de chaque individu et non de la condition physique. Les individus les mieux classés sont plus préférés que les moins bien classés.

Chromosome	Valeur de Fitness	Rang
A	8.1	1
B	8.0	4
C	8.05	2
D	7.95	6
E	8.02	3
F	7.99	5

En cela, nous supprimons le concept de valeur de fitness lors de la sélection d'un parent. Cependant, chaque individu de la population est classé en fonction de sa condition physique. La sélection des parents dépend du rang de chaque individu et non de la condition physique. Les individus les mieux classés sont plus préférés que les moins bien classés.

4.1.3 Sélection par Tournoi

La sélection de tournoi est une stratégie de sélection utilisée pour sélectionner les candidats les plus aptes de la génération actuelle dans un algorithme génétique. Ces candidats sélectionnés sont ensuite transmis à la génération suivante. Dans une sélection de tournoi K-way, nous sélectionnons k-individus et organisons un tournoi parmi eux. Seul le candidat le plus apte parmi les candidats sélectionnés est choisi et transmis à la génération suivante. De cette façon, de nombreux tournois de ce type ont lieu et nous avons notre sélection finale de candidats qui passent à la génération suivante. Il a également un paramètre appelé la pression de sélection qui est une mesure probabiliste de la probabilité qu'un candidat participe à un tournoi. Si la taille du tournoi est plus grande, les candidats faibles ont moins de chances d'être sélectionnés car ils doivent rivaliser avec un candidat plus fort. Le paramètre de pression de sélection détermine le taux de

convergence de l'AG. Plus la pression de sélection sera élevée, plus le taux de convergence sera élevé. Les GA sont capables d'identifier des solutions optimales ou quasi optimales sur une large gamme de pressions de sélection. La sélection de tournoi fonctionne également pour les valeurs de condition physique négatives.

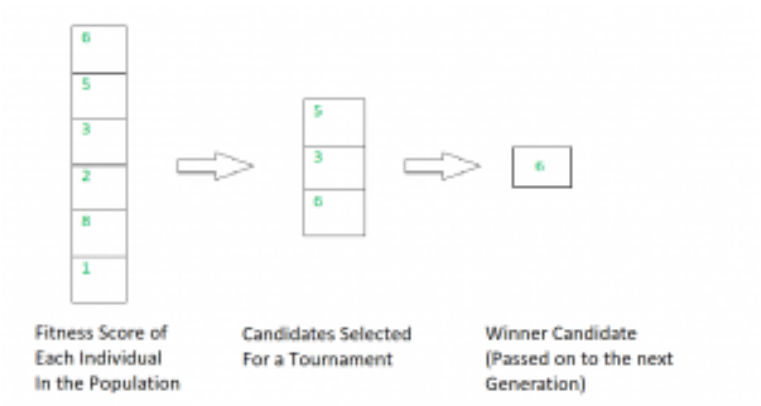


Figure 5: Sélection par Tournoie.

4.2 Le Croisement

L'opérateur de croisement est analogue à la reproduction et au croisement biologique. Dans ce cas, plus d'un parent est sélectionné et une ou plusieurs progénitures sont produites à l'aide du matériel génétique des parents. Le crossover est généralement appliqué dans un GA avec une probabilité élevée

4.2.1 croisement en un point

Dans ce croisement à un point, un point de croisement aléatoire est sélectionné et les queues de ses deux parents sont échangées pour obtenir de nouvelles progénitures.

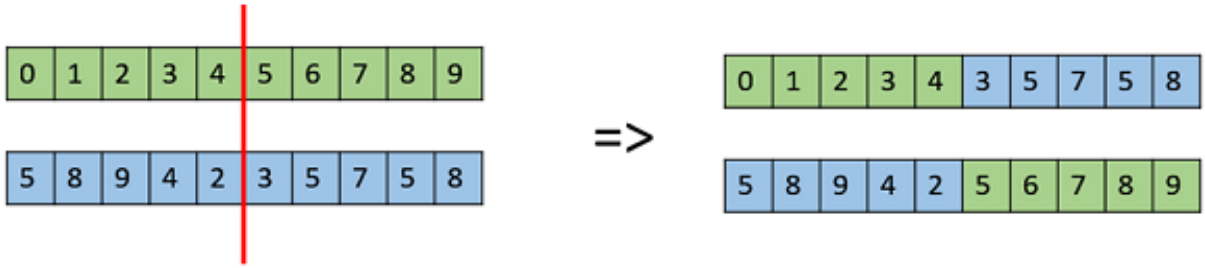


Figure 6: croisement en un point.

4.2.2 Croisement Multiple

Le croisement multipoint est une généralisation du croisement à un point dans lequel des segments alternés sont échangés pour obtenir de nouveaux descendants.

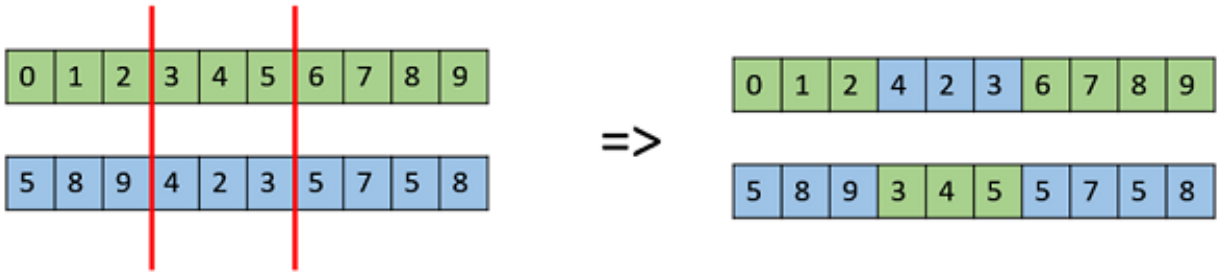


Figure 7: croisement Multiple.

4.2.3 Croisement Uniforme

Dans un croisement uniforme, nous ne divisons pas le chromosome en segments, nous traitons plutôt chaque gène séparément. En cela, nous lançons essentiellement une pièce pour chaque chromosome pour décider s'il sera inclus ou non dans la progéniture. Nous pouvons également biaiser la pièce vers un parent, pour avoir plus de matériel génétique chez l'enfant de ce parent.

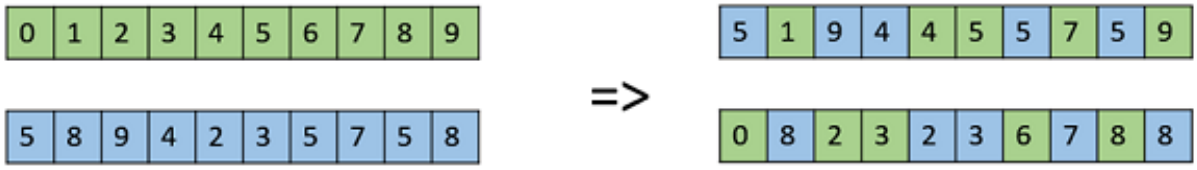


Figure 8: croisement Uniforme.

4.2.4 Croisement Arithmétique

Ceci est couramment utilisé pour les représentations entières et fonctionne en prenant la moyenne pondérée des deux parents en utilisant les formules suivantes: $child1 = \alpha x + (1 - \alpha y)$ $child2 = \alpha x + (1 - \alpha y)$

Évidemment, si $\alpha = 0.5$, alors les deux enfants seront identiques, comme le montre l'image suivante.

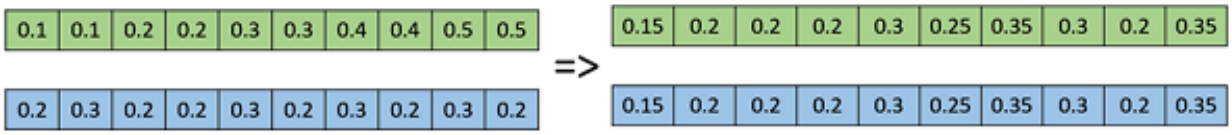


Figure 9: croisement Arithmétique.

4.3 La mutation

En termes simples, la mutation peut être définie comme un petit ajustement aléatoire dans le chromosome, pour obtenir une nouvelle solution. Il est utilisé pour maintenir et introduire la diversité dans la population génétique et est généralement appliqué avec une faible probabilité. Si la probabilité est très élevée, l'AG est réduit à une recherche aléatoire.

La mutation est la partie de l'AG qui est liée à "l'exploration" de l'espace de recherche. Il a été observé que la mutation est essentielle à la convergence de l'AG alors que le croisement ne l'est pas.

4.3.1 Bit Flip Mutation

Dans cette mutation de retournement de bit, nous sélectionnons un ou plusieurs bits aléatoires et les retournons. Ceci est utilisé pour les GA encodés en binaire.

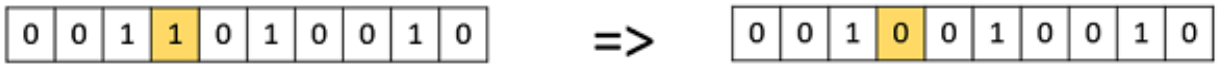


Figure 10: bit flip mutation.

4.3.2 La mutation de brouillage

La mutation de brouillage ou Scramble mutation est également populaire avec les représentations de permutation. En cela, à partir du chromosome entier, un sous-ensemble de gènes est choisi et leurs valeurs sont brouillées ou mélangées au hasard.

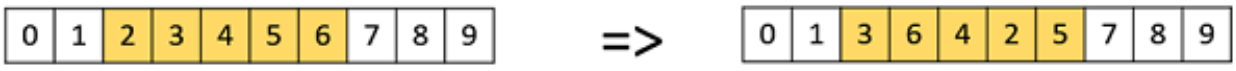


Figure 11: La mutation de brouillage.

4.3.3 La mutation d'inversion

Dans la mutation d'inversion, nous sélectionnons un sous-ensemble de gènes comme dans la mutation de brouillage, mais au lieu de mélanger le sous-ensemble, nous inversons simplement la chaîne entière dans le sous-ensemble.

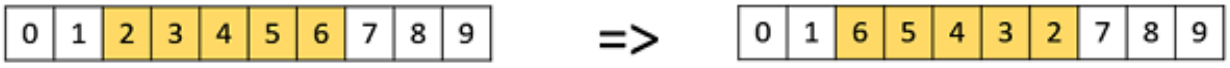


Figure 12: La mutation d'inversion.

5 L'algorithme

Les étapes impliquées dans le développement de l'algorithme génétique sont dans l'ordre suivant :

Initialisation

C'est la première étape dans laquelle une population de taille appropriée de chaînes binaires, de valeurs de bobine ou autre chose dépend du type de codage de la longueur de chromosome appropriée est créée. Toutes les chaînes sont évaluées pour leurs valeurs de fitness à l'aide de la fonction de fitness spécifiée. La fonction objectif est interprétée dans la légère minimisation et maximisation et devient la fonction de fitness.

Sélection

Cela implique la sélection des chromosomes de la population actuelle pour former un pool d'accouplement pour la production de la prochaine génération. La procédure de sélection est stochastique dans laquelle les chromosomes plus aptes ont une meilleure chance d'être sélectionnés.

Croisement

Cette étape aboutit à la création de deux chromosomes descendants à partir de chaque paire de parents sélectionnés au hasard. Les deux chromosomes parents sélectionnés sont coupés aux mêmes points de croisement sélectionnés au hasard pour obtenir deux sous-chaînes par chaîne parent. La deuxième sous-chaîne est ensuite mutuellement échangée et combinée avec la première sous-chaîne respective pour former deux chromosomes descendants.

Mutation

Parmi les membres de la population générés, aléatoirement autant d'éléments de la descendance sont mutés avec une probabilité égale à P_{mut}. Ceci est généralement très petit et évite la création de sous-espaces de recherche entièrement différents. Cela évite que la recherche GA ne devienne absolument aléatoire.

La nouvelle population subit le test de Fitness. Les étapes sont répétées et enfin, les valeurs des variables obtenues représentent la solution optimisée.

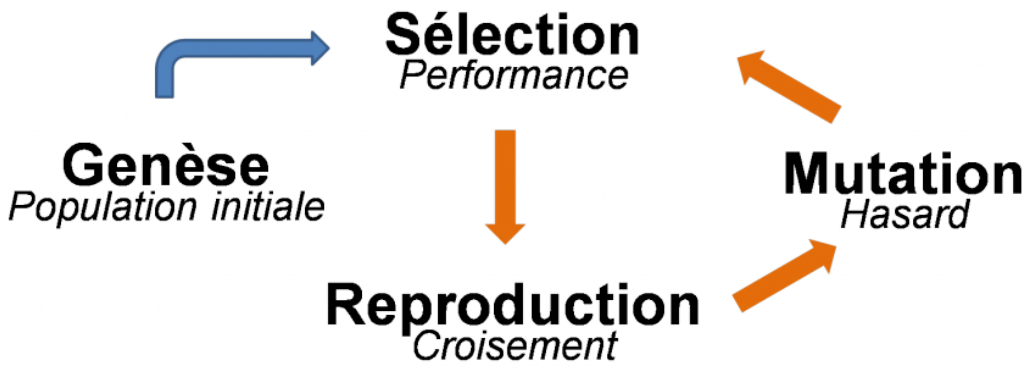


Figure 13: Méthode générale d'algorithme génétique.

$fonctionGA(){$
$populationinitiale;$
$calculerFitness;$

$while(valeurduFitness! = criteresderealisation){$
$Selection;$

$Croisement;$

$Mutation;$

$CalculerFitness;$
}
}

6 L'implémentassions du GA sous Python

6.1 Problème du Sac à dos

6.1.1 Définition

Le problème du sac à dos est un problème d'optimisation combinatoire: étant donné un ensemble d'éléments, chacun avec un poids et une valeur, déterminer le nombre de chaque élément à inclure dans une collection de sorte que le poids total soit inférieur ou égal à une limite donnée et la valeur totale est aussi grande que possible. Il tire son nom du problème rencontré par quelqu'un qui est contraint par un sac à dos de taille fixe et doit le remplir avec les objets les plus précieux. Le problème se pose souvent dans l'allocation des ressources où les décideurs doivent choisir parmi un ensemble de projets ou de tâches non divisibles sous un budget fixe ou une contrainte de temps, respectivement.

6.1.2 résoudre le problème du sac à dos à l'aide d'algorithmes génétiques en Python

tout d'abord, nous importons les bibliothèques requises

```
import numpy as np
import random
```

ensuite nous définissons les paramètres

```
n_objects = 20
max_weight = 3
n_population = 100
mutation_rate = 0.3
```

Génération d'une liste de différents objets avec leurs poids et leurs valeurs

```
weight_value = [[x,y] for x,y in zip(np.random.randint(0,10,n_objects)/10,
                                     np.random.randint(0,100,n_objects))]
object_list = np.array(['Water', 'Food', 'Pants', 'Socks', 'Boots',
'Shirts', 'Coat', 'Blanket', 'Laptop', 'TV', 'Cellphone', 'Book',
'Gloves', 'Towel', 'Sunscream', 'Glasses', 'Fork', 'Knife', 'Matches',
'Chair'])
objects_dict = { x:y for x,y in zip(object_list,weight_value)}

def get_current_weight_value(objects_list, objects_dict):
    weight = 0
    value = 0
    for o in objects_list:
        o = objects_dict[o]
        weight += o[0]
        value += o[1]
    return weight, value


def fit_in_knapsack(objects_list, max_weight, objects_dict):
    r = []
    for o in objects_list:
        r.append(o)
        weight, value = get_current_weight_value(r, objects_dict)
        if weight > max_weight:
            r.pop()
```

```
            return r
    return r

print(objects_dict)
```

Output 'Water': [0.8, 74], 'Food': [0.2, 26], 'Pants': [0.7, 74], 'Socks': [0.6, 53], 'Boots': [0.1, 4], 'Shirts': [0.6, 5], 'Coat': [0.9, 87], 'Blanket': [0.9, 20], 'Laptop': [0.4, 68], 'TV': [0.9, 92], 'Cellphone': [0.1, 82], 'Book': [0.6, 20], 'Gloves': [0.5, 2], 'Towel': [0.4, 95], 'Sunscream': [0.4, 66], 'Glasses': [0.5, 74], 'Fork': [0.7, 73], 'Knife': [0.5, 21], 'Matches': [0.5, 64], 'Chair': [0.3, 98]

1.Initialisation(Créer le premier ensemble de population)

```
def fit_in_knapsack(objects_list, max_weight, objects_dict):
    r = []
    for o in objects_list:
        r.append(o)
        weight, value = get_current_weight_value(r, objects_dict)
        if weight > max_weight:
            r.pop()
            return r
    return r

def genesis(object_list, n_population, max_weight, objects_dict):

    population_set = []
    for i in range(n_population):
        #Randomly generating a new solution
        sol_i = object_list[np.random.choice(list(range(n_objects)),
        n_objects, replace=False)]
        sol_i = fit_in_knapsack(sol_i, max_weight, objects_dict)
        population_set.append(sol_i)
    return np.array(population_set)

population_set = genesis(object_list, n_population, max_weight, objects_dict)
population_set[:5]
```

Output array([list(['Chair', 'Fork', 'Shirts', 'Boots', 'Cellphone', 'Food', 'Laptop', 'Sunscream']), list(['Boots', 'Socks', 'Food', 'Shirts', 'Sunscream', 'Water', 'Chair']), list(['Book', 'Socks', 'Coat', 'Matches', 'Laptop']), list(['Shirts', 'Pants', 'Blanket', 'Boots', 'Food']), list(['Sunscream', 'Towel', 'Coat', 'Cellphone', 'TV'])])

2.Évaluer Fitness des solutions

```
def get_all_fitnes(population_set, objects_dict):
    fitnes_list = np.zeros(n_population)

    #Looping over all solutions computing the fitness for each solution
    for i in  range(n_population):
        _, fitnes_list[i] =
        get_current_weight_value(population_set[i], objects_dict)
```

```
    return fitnes_list

fitnes_list = get_all_fitnes(population_set,objects_dict)
fitnes_list[:10]
```

Output array([422., 326., 292., 129., 422., 307., 313., 271., 259., 344., 268.]

3.Selection(Par Roulette)

```
def progenitor_selection(population_set,fitnes_list):
    total_fit = fitnes_list.sum()
    prob_list = fitnes_list/total_fit

    #Notice there is the chance that a progenitor. mates with oneself
    progenitor_list_a =
    np.random.choice(list(range(len(population_set))),
    len(population_set),p=prob_list, replace=True)
    progenitor_list_b =
    np.random.choice(list(range(len(population_set))),
    len(population_set),p=prob_list, replace=True)

    progenitor_list_a = population_set[progenitor_list_a]
    progenitor_list_b = population_set[progenitor_list_b]


    return np.array([progenitor_list_a,progenitor_list_b])


progenitor_list = progenitor_selection(population_set,fitnes_list)
progenitor_list[0][2]
```

Output ['Shirts', 'Cellphone', 'Sunscream', 'Book', 'Fork', 'Towel']

4.Croisement

```
def mate_progenitors(prog_a, prog_b, max_weight, objects_dict):
    offspring = []

    for i in zip(prog_a, prog_b):
        offspring.extend(i)
        offspring = list(dict.fromkeys(offspring)) #Removing duplicates
        offspring = fit_in_knapsack(offspring, max_weight, objects_dict)

    return offspring



def mate_population(progenitor_list, max_weight, objects_dict):
    new_population_set = []
    for i in range(progenitor_list.shape[1]):
```

```
        prog_a, prog_b = progenitor_list[0][i], progenitor_list[1][i]
        offspring = mate_progenitors(prog_a, prog_b, max_weight, objects_dict)
        new_population_set.append(offspring)

    return new_population_set

new_population_set = mate_population(progenitor_list, max_weight, objects_dict)
new_population_set[0]
```

Output ['Shirts', 'Matches', 'Cellphone', 'Boots', 'Coat', 'Food']

5.Mutation

```
def mutate_offspring(offspring, max_weight, object_list, objects_dict):
    for mutation_number in range(int(len(offspring)*mutation_rate)):

        a = np.random.randint(0,len(object_list))
        b = np.random.randint(0,len(offspring))

        offspring.insert(b, object_list[a])

    offspring = fit_in_knapsack(offspring, max_weight, objects_dict)

    return offspring


def mutate_population(new_population_set, max_weight,
object_list, objects_dict):
    mutated_pop = []
    for offspring in new_population_set:
        mutated_pop.append(mutate_offspring(offspring, max_weight,
        object_list, objects_dict))
    return mutated_pop

mutated_pop = mutate_population(new_population_set,
max_weight,object_list, objects_dict)
mutated_pop[0]
```

Output ['Shirts', 'Matches', 'Matches', 'Cellphone', 'Boots', 'Coat', 'Food']

7.Condition d'arête

```
best_solution = [-1,-np.inf,np.array([])]
for i in range(10000):

    #Saving the best solution
    if fitnes_list.max() > best_solution[1]:
        best_solution[0] = i
        best_solution[1] = fitnes_list.max()
        best_solution[2] = np.array(mutated_pop)[fitnes_list.max()
```

```
        == fitnes_list]

    progenitor_list = progenitor_selection(population_set,fitnes_list)
    new_population_set = mate_population(progenitor_list,
    max_weight, objects_dict)

    mutated_pop = mutate_population(new_population_set, max_weight,object_list, objects_dic
get_current_weight_value(best_solution[2][0], objects_dict)
```

Output (2.9000000000000004, 715)

6.2 Résolution des problèmes d'optimisation aves les AG

6.2.1 Maximisation d'une fonction a deux variables

Le problème que nous allons essayer de résoudre ici est de trouver le maximum d'une fonction 3D semblable à un chapeau. Il est défini comme $f(x, y) = \sin(\sqrt{(x^2 + y^2)})$. Nous limiterons notre problème aux bornes de $4 > x > -4$ et $4 > y > -4$

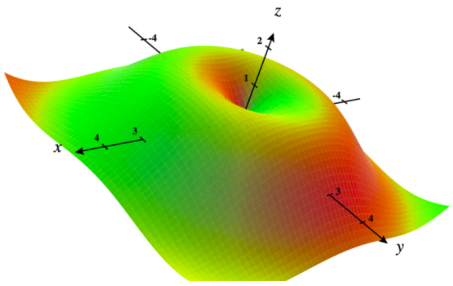


Figure 14: Graph de $\sin(\sqrt{(x^2 + y^2)})$

La première étape consiste à générer notre population initiale. Nous allons itérer sur plusieurs générations en l'améliorant jusqu'à ce que nous trouvions une solution acceptable. La première génération est générée aléatoirement.

```
import numpy as np
def generate_population(size, x_boundaries, y_boundaries):
    lower_x_boundary, upper_x_boundary = x_boundaries
    lower_y_boundary, upper_y_boundary = y_boundaries

    population = []
    for i in range(size):
        individual = {
            "x": random.uniform(lower_x_boundary, upper_x_boundary),
            "y": random.uniform(lower_y_boundary, upper_y_boundary),
        }
        population.append(individual)

    return population
```

Notre fonction de genèse attend trois arguments : le nombre d'individus que la population devrait avoir, un tuple indiquant les limites sur l'axe des x et un tuple indiquant les limites sur l'axe des y, de sorte que nos

individus s'adaptent aléatoirement à ces limites.
Maintenant, définissons notre fonction de fitness. Ce sera notre évaluateur, qui exprimera à quel point un individu est meilleur ou pire l'un par rapport à l'autre. Les individus avec la meilleure fitness devraient être préservés et se reproduire tandis que les pires devraient tomber, comme dans la nature. Dans notre cas, comment nous voulons trouver notre fonction maximale, nous pouvons simplement appliquer notre fonction objectif à un individu et les plus grands nombres seront également la plus grande forme physique. Si nous voulions trouver le minimum, la forme physique pourrait être exprimée comme le résultat de la fonction fois -1, de sorte que des valeurs plus petites deviennent une plus grande fitness.

```
import math

def apply_function(individual):
    x = individual["x"]
    y = individual["y"]
    return math.sin(math.sqrt(x ** 2 + y ** 2))
```

Pour sélectionner les individus à reproduire nous utiliserons la méthode de la roulette.

```
def choice_by_roulette(sorted_population, fitness_sum):
    offset = 0
    normalized_fitness_sum = fitness_sum

    lowest_fitness = apply_function(sorted_population[0])
    if lowest_fitness < 0:
        offset = -lowest_fitness
        normalized_fitness_sum += offset * len(sorted_population)

    draw = random.uniform(0, 1)

    accumulated = 0
    for individual in sorted_population:
        fitness = apply_function(individual) + offset
        probability = fitness / normalized_fitness_sum
        accumulated += probability

        if draw <= accumulated:
            return individual
```

Peuplant alors la prochaine génération. Il devrait avoir la même longueur que le premier, nous allons donc itérer 10 fois en sélectionnant deux individus chacun en utilisant notre roulette puis en les croisant. L'individu qui en résulte recevra une perturbation mineure (mutation), nous ne nous en tenons donc pas à la zone de confort et recherchons des solutions encore meilleures que celles que nous avons jusqu'à présent.
Il existe plusieurs techniques de croisement pour les nombres réels, pour des raisons de simplicité, utilisons la moyenne arithmétique.
Pour la mutation, il existe également de nombreuses options - nous additionnerons simplement un petit nombre aléatoire entre un intervalle fixe. Cet intervalle est le taux de mutation et peut être ajusté en conséquence, utilisons [-0,05, 0,05]. Pour des espaces de recherche plus grands, vous pouvez choisir des intervalles plus grands et les diminuer de génération en génération.

```
def sort_population_by_fitness(population):
    return sorted(population, key=apply_function)


def crossover(individual_a, individual_b):
```

```
    xa = individual_a["x"]
    ya = individual_a["y"]

    xb = individual_b["x"]
    yb = individual_b["y"]

    return {"x": (xa + xb) / 2, "y": (ya + yb) / 2}


def mutate(individual):
    next_x = individual["x"] + random.uniform(-0.05, 0.05)
    next_y = individual["y"] + random.uniform(-0.05, 0.05)

    lower_boundary, upper_boundary = (-4, 4)

    # Guarantee we keep inside boundaries
    next_x = min(max(next_x, lower_boundary), upper_boundary)
    next_y = min(max(next_y, lower_boundary), upper_boundary)

    return {"x": next_x, "y": next_y}


def make_next_generation(previous_population):
    next_generation = []
    sorted_by_fitness_population =
    sort_population_by_fitness(previous_population)
    population_size = len(previous_population)
    fitness_sum = sum(apply_function(individual)
    for individual in population)

    for i in range(population_size):
        first_choice =
        choice_by_roulette(sorted_by_fitness_population,fitness_sum)
        second_choice =
        choice_by_roulette(sorted_by_fitness_population, fitness_sum)

        individual = crossover(first_choice, second_choice)
        individual = mutate(individual)
        next_generation.append(individual)

    return next_generation
```

Alors c'est ça! Nous avons maintenant les trois étapes d'un AG : sélection, croisement et mutation. Notre méthode principale est alors simplement comme ça.

```
generations = 100

population = generate_population(size=10, x_boundaries=(-4, 4),
y_boundaries=(-4, 4))

i = 1
while True:
    print(f"GENERATION_{i}")
```

```
    for individual in population:
        print(individual, apply_function(individual))

    if i == generations:
        break

    i += 1

    population = make_next_generation(population)

best_individual = sort_population_by_fitness(population)[-1]
print("\nFINAL_RESULT")
print(best_individual, apply_function(best_individual))
```

La variable best individual tiendra notre individu avec la meilleure forme physique après ces 100 générations. Cela peut être la solution optimale exacte ou non. Voyons les dernières lignes de sortie pour une exécution expérimentale (notez qu'en raison des paramètres aléatoires, nous obtiendrons très probablement des résultats différents mais similaires): **Output**

GENERATION 100 'x': -1.0665224807251312, 'y': -1.445963268888755 0.9745828000809058 'x': -1.0753606354537244, 'y': -1.4293367491155182 0.976355423070003 'x': -1.0580786664161246, 'y': -1.3693549033564183 0.9872729309456848 'x': -1.093601208942564, 'y': -1.383292089777704 0.9815156357267611 'x': -1.0464963866796362, 'y': -1.3461172606906064 0.9910018621648693 'x': -0.987226479369966, 'y': -1.4569537217049857 0.9821687265560713 'x': -1.0501568673329658, 'y': -1.430577408679398 0.9792937786319258 'x': -1.0291192465186982, 'y': -1.4289167102720242 0.9819781801342095 'x': -1.098502968808768, 'y': -1.3738230550364259 0.9823409690311633 'x': -1.091317403073779, 'y': -1.4256574643591997 0.9748817266026281
FINAL RESULT 'x': -1.0464963866796362, 'y': -1.3461172606906064 0.9910018621648693

Notre résultat final était très proche de l'une des solutions possibles (cette fonction a plusieurs maximums à l'intérieur de nos limites, qui est de 1,0 comme vous pouvez le voir sur le tracé au début). Notez que nous avons utilisé les techniques les moins sophistiquées possibles, donc ce résultat est en quelque sorte attendu - c'est un point de départ pour affiner jusqu'à ce que nous soyons en mesure de trouver de meilleures solutions avec moins de générations.

6.2.2 Résolution d'un Problème d'optimisation sous contraint

Les GA tentent de faire évoluer la population de chromosomes plus aptes en appliquant trois opérateurs évolutifs clés: la sélection, le croisement et la mutation. L'objectif est de produire une nouvelle génération ou des descendants avec une meilleure valeur de fitness que leurs parents. La plupart des problèmes de conception d'optimisation d'ingénierie sont difficiles à résoudre à l'aide d'algorithmes conventionnels car ils comprennent des contraintes spécifiques au problème (linéaire, non linéaire, égalité ou inégalité). Malgré le succès de GA dans un large éventail d'applications, la résolution de problèmes d'optimisation sous contrainte n'est pas une tâche facile. La technique la plus courante consiste à appliquer des **fonctions de pénalité**. En conséquence, le problème est converti d'un problème d'optimisation contraint à un problème d'optimisation non contraint. L'inconvénient majeur de ces fonctions de pénalité est l'exigence d'une définition et d'un réglage correct de leurs paramètres, ce qui peut être difficile et problématique.
Ici, nous allons minimiser la fonction continue suivante appelée **Fonction de Himmelblau**.

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

Nous allons borner ou limiter les valeurs des deux variables x et y entre $[-6, 6]$. Les limites peuvent être différentes sur les deux.

Dans cette domaine de x et y, nous avons le *maximum* suivant à

$$f(-0.270845, -0.923039) = 181.617$$

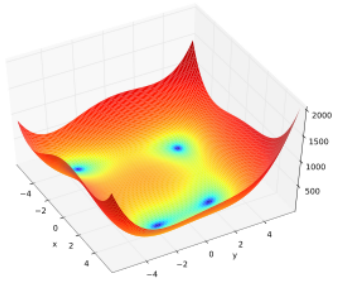


Figure 15: Graph de la fonction de Himmelblau

et suivant quatre minimums identiques,

$$f(3.0, 2.0) = 0.0$$
$$f(-2.805118, 3.131312) = 0.0$$
$$f(-3.779310, -3.283186) = 0.0$$
$$f(3.584428, -1.848126) = 0.0$$

Les algorithmes évolutionnaires sont généralement des procédures d'optimisation sans contraintes. Puisque l'optimisation contrainte est un scénario plus réel, nous allons également mettre une contrainte sur les variables de sorte que leur somme soit inférieure à zéro.

$$x + y < 0$$

Cela nous laisse les valeurs optimales attendues pour x et y à
$x = -3.779310$ et $y = -3.283186$ $f(x, y) = 0.0$ **fonction de pénalité** est la fonction de pénalité qui renvoie une valeur de pénalité qui est ajoutée à la valeur du fitness (pour la minimisation) car cela rendra l'individu moins en forme (une valeur de forme physique élevée est moins adaptée aux problèmes de minimisation, à l'opposé de la maximisation). Généralement, la pénalité est proportionnelle à la mesure dans laquelle l'individu viole la contrainte. nous pouvons utiliser exactement le même code ci-dessus pour la maximisation de la fonction à deux variables, la seule modification consiste à définir une fonction de pénalité, puis à modifier la fonction de fitness en ajoutant la valeur de pénalité pour la minimisation ou en la soustrayant pour la maximisation.

```
def penalty_function(individual):
    x=individual['x']
    y=individual['y']
    return sum(x+y)**2

def fitness_function(individual):
    x = individual["x"]
    y = individual["y"]
    return math.sin(math.sqrt(x ** 2 + y ** 2))+
```

6.3 Une interface Graphique avec TKINTER

Afin de faciliter l'utilisation d'algorithmes génétiques pour les problèmes d'optimisation par les non-informaticiens et les utilisateurs réguliers, nous avons créé une interface graphique utilisant Tkinter.

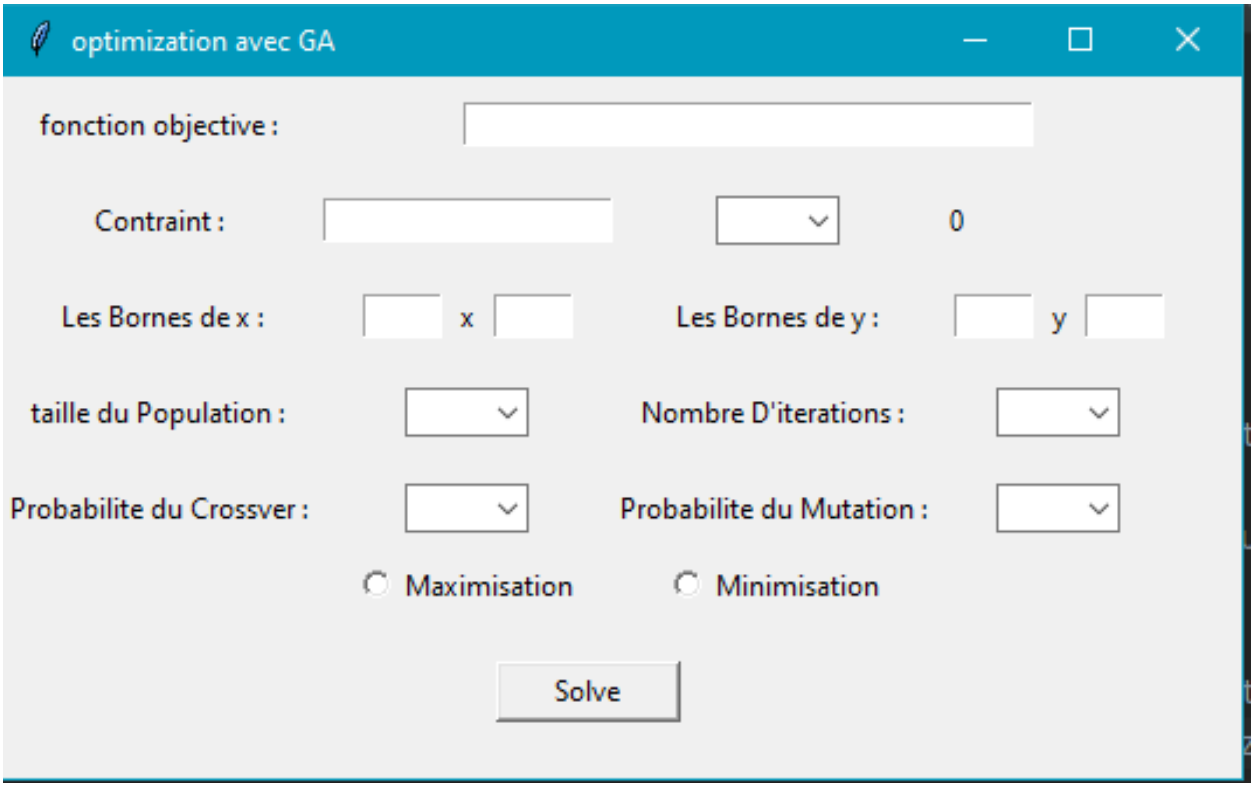
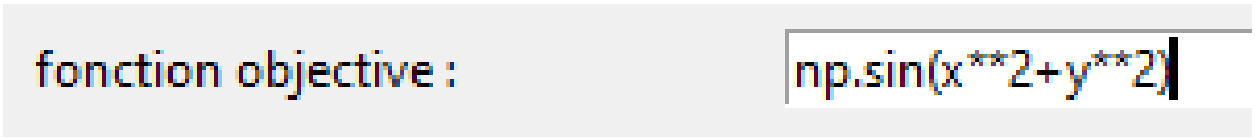


Figure 16: L'interface Graphique

Tkinter est la bibliothèque graphique standard pour Python. Python lorsqu'il est combiné avec Tkinter fournit un moyen rapide et facile de créer des applications GUI. Tkinter fournit une puissante interface orientée objet à la boîte à outils Tk GUI.

L'interface graphique que nous avons créée consiste de:
1. l'entrée de la fonction objectif (obligatoire)
où vous devez taper votre fonction objectif en suivant les règles de python (* pour la multiplication, ** pour la puissance et sin, exp, log , etc. doivent être implémentés en utilisant numpy sous la forme np.sin, np.exp, np.log)

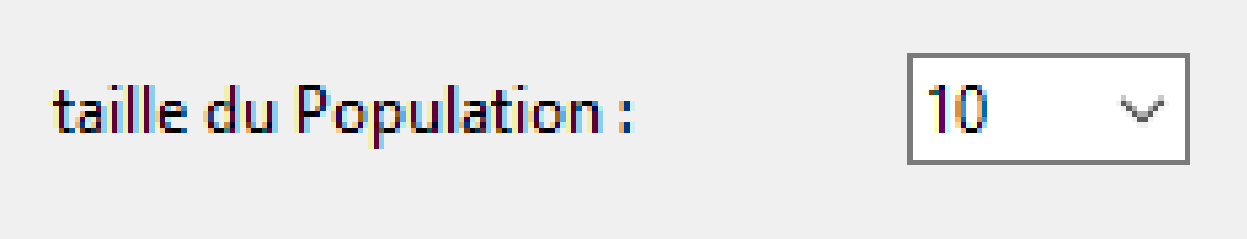


2. l'entrée de l'équation de contrainte (pas obligatoire)
où vous devez taper l'équation de votre contrainte en passant tous les termes d'un côté, seul 0 doit rester de l'autre côté, à côté de l'entrée, il y a une sélection pour le type d'équation de contrainte (égalité ou inégalité).

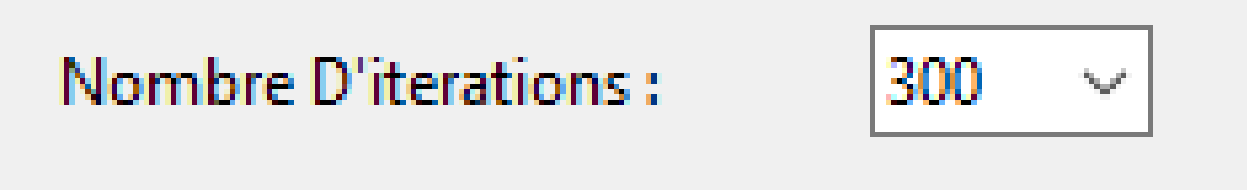
3. quatre entrées pour les bornes de X et Y. (pas obligatoire)

puis il y a les paramètres liés à l'algorithme génétique :

1. une entrée pour la taille de la population:(obligatoire)
où l'utilisateur doit choisir la taille de la population qu'il souhaite utiliser (il doit s'agir d'un nombre entier).

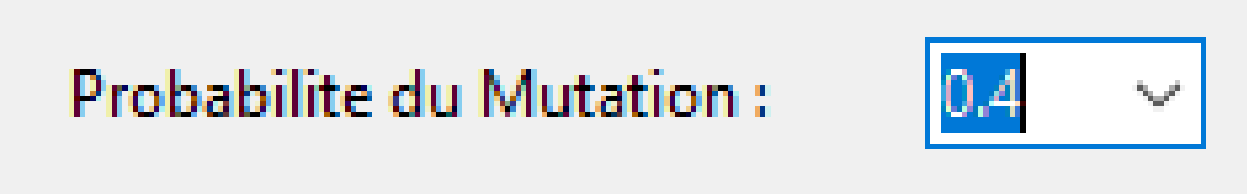


2. une entrée pour le nombre d'itérations: (obligatoire)
autrement dit le nombre de générations, où l'utilisateur doit choisir le nombre de générations auquel l'algorithme s'arrête (il doit également être un entier)

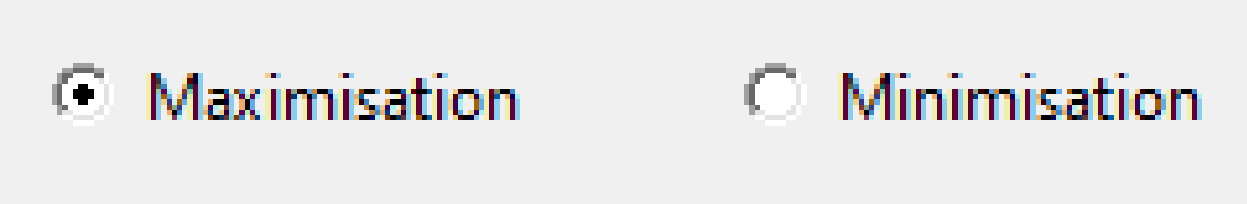


3. L'entrée de probabilité de croisement: (obligatoire)
où l'utilisateur doit choisir la probabilité de croiser les individus.

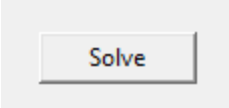
4. L'entrée pour la probabilité de mutation: (obligatoire) où l'utilisateur choisit la probabilité de mutation des individus.



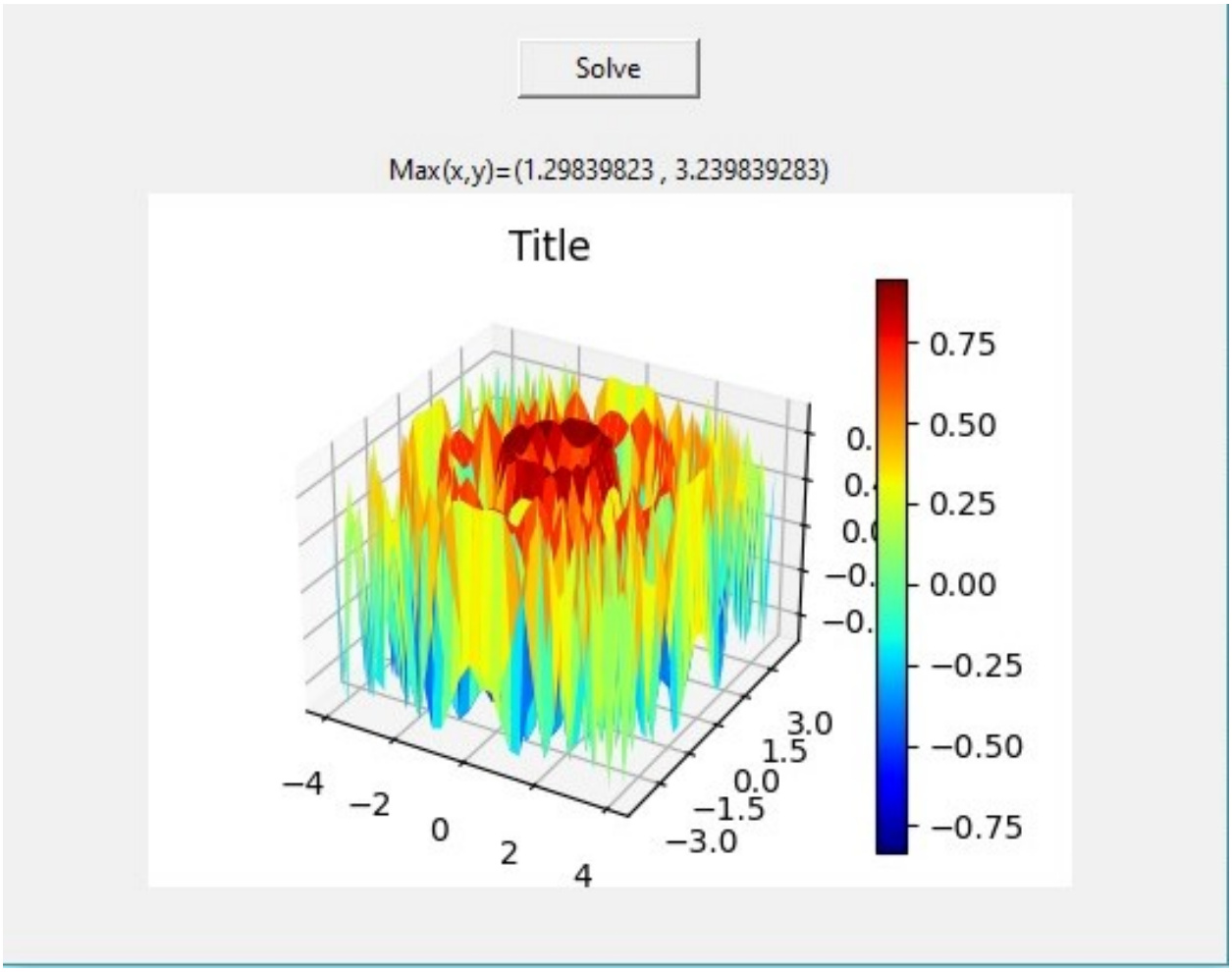
enfin il y a une RadioBox où l'utilisateur doit choisir s'il veut une maximisation ou une minimisation pour le problème d'optimisation.



après avoir rempli toutes les entrées requises, l'utilisateur doit appuyer sur le bouton de résolution.



après avoir récupéré les données et appliqué l'algorithme génétique sur le problème d'optimisation, juste sous le bouton de résolution, une solution x-y apparaîtra avec un graphe 3D pour la fonction objectif (vous pouvez déplacer le graphe 3D en cliquant dessus et en déplaçant la souris)



sur le côté droit, une case avec la liste des générations que nous traversons jusqu'à ce que le résultat final apparaisse, dans chaque génération, il y a une liste d'individus et dans chaque individu, il y a une solution x-y.

{'x': 0.7992508848133295, 'y': 0.8100731789737674}, 0.9622142433733053)
{'x': 0.8194170095928314, 'y': 0.757216094466858}, 0.9473386689744104)
{'x': 0.8361033611368691, 'y': 0.8302551539605691}, 0.9834104857500957)
{'x': 0.8236024358497244, 'y': 0.8487849517380489}, 0.9852376767691818)
{'x': 0.8282485012821486, 'y': 0.7523517958607172}, 0.9496223933469747)
{'x': 0.845221733835709, 'y': 0.7425900011241083}, 0.9538599730805807)
{'x': 0.8310562244116214, 'y': 0.779149432320074}, 0.9629480188019243)
{'x': 0.7824212183614403, 'y': 0.8160624373974277}, 0.9574811625863958)
GENERATION 300

{'x': 0.8658200082969978, 'y': 0.8455463726981485}, 0.994365720739928)
{'x': 0.7844001804574288, 'y': 0.8420195838258666}, 0.9697685671093755)
{'x': 0.7561092111426672, 'y': 0.82414893087173}, 0.9492751316637522)
{'x': 0.8476915384501197, 'y': 0.8237807496655086}, 0.9849692113718288)
{'x': 0.8175030189760234, 'y': 0.7927574979166472}, 0.9626906646410579)
{'x': 0.8155073942782933, 'y': 0.76996642072285}, 0.9514462116281801)
{'x': 0.8699171906976384, 'y': 0.823539131954287}, 0.990790132175589)
{'x': 0.7710929994350997, 'y': 0.7533593078834455}, 0.9176534736325557)
{'x': 0.7752563584012161, 'y': 0.7850184164121832}, 0.9381598984228854)
{'x': 0.8261614721885929, 'y': 0.8271886339318215}, 0.979261527143884)

FINAL RESULT
{'x': 0.8658200082969978, 'y': 0.8455463726981485}, 0.994365720739928

Conclusion