Université Abdelmalek Essaidi

Ecole National des Science Appliquées

De Al Hoceima

Filière: Génie informatique

Titre

Résolution du problème du sac à dos avec l'algorithme génétique

Réalisé par :

Mohamed MIRI

Année Universitaire : 2019/2020

LinkedIn : https://www.linkedin.com/in/mohamed-miri-37299a163/
Email : miri.mohamed.1998@hotmail.com

Sommaire

Introduction

Le succès d'un projet ou le développement d'une entreprise est dans de nombreux cas liés à la capacité des ingénieurs et des décideurs de résoudre des problèmes de type : minimiser les coûts et maximiser la production avec l'inclusion d'un certain nombre de paramètres. Ce type de problème est appelé problème d'optimisation, où nous voulons minimiser une fonction de coût ou maximiser une fonction objective.

Les problèmes d'optimisation apparaissent dans plusieurs domaines, telle que la conception de systèmes mécaniques, traitement d'image, l'électronique et la recherche opérationnelle. Résoudre un tel problème est de donner les bonnes valeurs des paramètres pour avoir une solution optimale. Ces valeurs doivent respecter les contraintes liées au problème.

Dans ce travail, nous réalisons un projet qui entre dans le cadre de l'intelligence artificielle. Il s'agit d'une solution basée sur l'algorithme génétique pour trouver les objets à inclure pour le problème du sac à dos. Ceci a nécessité un codage réel, une fitness, une sélection proportionnelle à la valeur de la solution choisie, et des opérateurs de croisement, mutation convenable.

Chapitre 1 : Problème de sac à dos

Dans ce chapitre nous allons donner un aperçu historique du problème du sac à dos, ensuite, nous allons définir le problème et sa formulation mathématique, puis quelque algorithme utiliser dans la résolution de ce problème, enfin nous allons citer quelques domaines d'application dans la vie réelle.

I. Historique :

Le problème du sac à dos est l'un des 21 problèmes NP-complets de Richard Karp, exposés dans son article de 1972. Il est intensivement étudié depuis le milieu du XX^{e} siècle. La formulation du problème est fort simple, mais sa résolution est plus complexe.

II. Principe :

On dispose d'un sac pouvant supporter un poids maximal donné et de divers objets ayant chacun une valeur et un poids. Il s'agit de choisir les objets à emporter dans le sac afin d'obtenir la valeur totale la plus grande tout en respectant la contrainte du poids maximal. C'est un problème d'optimisation avec contrainte.

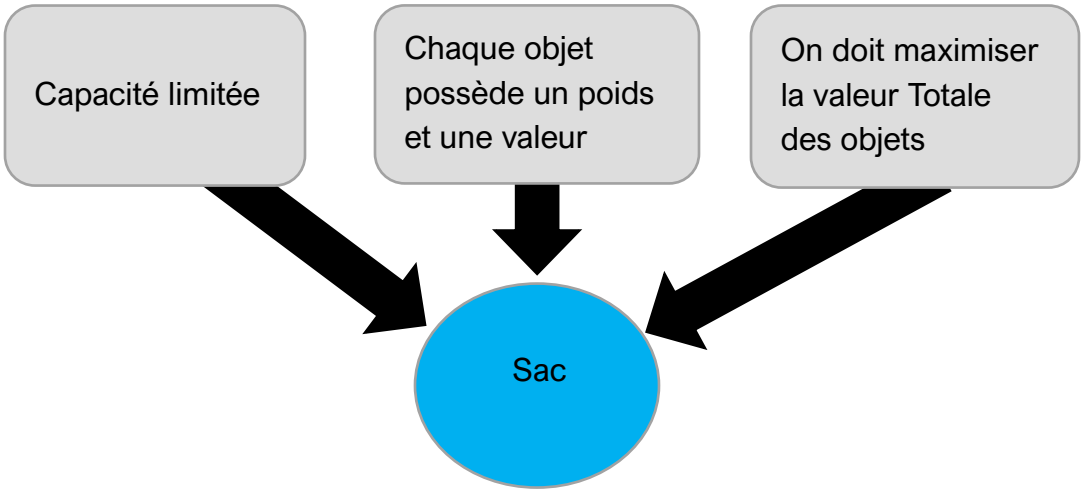


Figure 1 : Le problème sac à dos 1

Ce problème peut se résoudre par force brute(naïve), c'est-à-dire en testant tous les cas possibles. Mais ce type de résolution présente un problème d'efficacité. Son coût en fonction du nombre d'objets disponibles croît de manière exponentielle.

III. Formulation mathématique:

Les données du problème peuvent être exprimées en termes mathématiques. Les objets sont numérotés par l'indice i variant de 1 à n.
Les nombres w_i et v_i représentent respectivement le poids et la valeur de l'objet numéro i. La capacité du sac sera notée W.

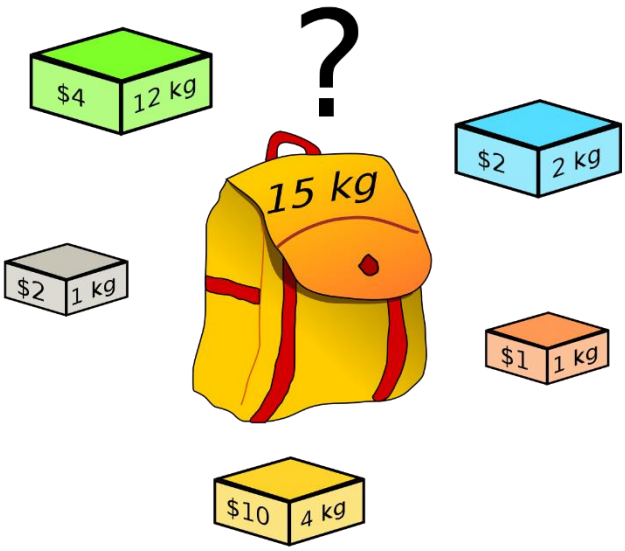


Figure 2 : Le problème de sac à dos 2

Modélisation :

Il existe de multiples façons de remplir le sac à dos. Pour décrire l'une d'elles il faut indiquer pour chaque élément s'il est pris ou non. On peut utiliser un codage binaire : l'état du i -ème élément vaudra $x_i = 1$ si l'élément est mis dans le sac, où $x_i = 0$ s'il est laissé de côté. Une façon de remplir le sac est donc complètement décrite par un vecteur, appelé vecteur contenu, ou simplement contenu : $X = [x_1, x_2, \ldots, x_n]$; et le poids associé, ainsi que la valeur associée, à ce remplissage, peuvent alors être exprimés comme fonction du vecteur contenu.

Exemple :

Si on dispose de 5 objets $X = \lfloor x_1, x_2, x_3, x_4, x_5 \rfloor$, et on veut mettre L'objet 1,2 et 4 dans le sac à dos, la formule sera : $S_1 = \lfloor 1,1,0,1,0 \rfloor$.

Formule :

Pour un contenu X donné, la valeur totale contenue dans le sac est naturellement :

$$z(X) = \sum_{i=1}^{n} x_i v_i$$

De même, la somme des poids des objets choisis est :

$$w(X) = \sum_{i=1}^{n} x_i w_i$$

Le problème peut alors être reformulé comme la recherche d'un vecteur contenu $X = \lfloor x_1, x_2, x_3, x_4, x_5 \rfloor$ (les composantes valant 0 ou 1), réalisant le maximum de la fonction valeur totale $z(X)$, sous la contrainte : $w(X) \leq W$

C'est-à-dire que la somme des poids des objets choisis ne dépasse pas la capacité du sac à dos.

Avec :

$v_i > 0, \forall i \in \{1, \dots, n\}$: tout objet a une valeur et apporte un gain ;

$w_i > 0, \forall i \in \{1, \dots, n\}$: tout objet a un certain poids et consomme des ressources.

Terminologie :

- $\mathrm{z(X)}$ est appelée fonction objective ;
- Tout vecteur X vérifiant la contrainte du poids est dit réalisable ;
- Si la valeur de $\mathrm{z(X)}$ est maximale, alors X est dit optimal.

IV. Résolution :

Il existe deux types de résolution :

1. Résolution approchée :

Comme pour la plupart des problèmes NP-complets, il peut être intéressant de trouver des solutions réalisables mais non optimales.

Algorithme utilisé : Algorithme glouton, Algorithme génétique, Algorithme basés sur les colonies de fourmis …

2. Résolution exacte :

Le problème du sac à dos, dans sa version classique, a été étudié en profondeur. Il existe donc de nombreuses méthodes exactes pour le résoudre.

Les méthodes : Programmation dynamique, Procédure de séparation et d'évaluation, Approches hybrides …

Et bien sûr la méthode naïve (force brute) qu'on va prendre comme référence dans cette recherche.

V. Analyse de la méthode naïve (force brute) :

La méthode naïve trouve toutes les solutions possibles, les tester puis trouve la meilleure, comme on sait les solutions d'un problème sont des vecteurs de longueurs n, où n est le nombre d'objet, et les composantes des vecteurs valant 0 ou 1.
On peut conclure que le nombre de solution est : 2^n
Et si on suppose que C est la complexité du calcule de la valeur d'une seule solution, la complexité de cette méthode sera : $C * 2^n = O(2^n)$

Pour 77 objets, Le nombre de combinaisons est : $2^{77} = 1.5 * 10^{23}$
Si on suppose que la complexité C s'exécute dans 10^{-6} seconde, le temps d'exécution du code complet pour 77 objets sera :

$$1.5 * 10^{23} * 10^{-6} = 1.5 * 10^{17} \text{ sec} = \ 154\ 153\ 413\ \ annéees$$

VI. Domaine d'application :

Puisque nous nous intéressons au développement des méthodes algorithmiques pour la résolution de KSP dans le but de faciliter quelques types de tâches dans la vie réelle et surtout dans la cryptographie, système d'aide à la gestion de portefeuille, l'optimisation d'un navire de charge, le découpage des matériaux …

Chapitre 2 : Algorithme génétique

Les algorithmes génétiques sont des algorithmes d'optimisation s'appuyant sur des techniques dérivées de la génétique et de l'évolution naturelle : croisements, mutations, sélection, etc.

I. Analogie avec la biologie :

1. Historique :

Les organismes vivants sont constitués de cellules, dont les noyaux comportent des chromosomes qui sont des chaînes d'ADN. L'élément de base de ces chaînes est un nucléotide, identifié par sa base azotée (A, T, C ou G). Sur chacun de ces chromosomes, une suite de nucléotides constitue une chaîne qui code les fonctionnalités de l'organisme (la couleur des yeux par exemple).

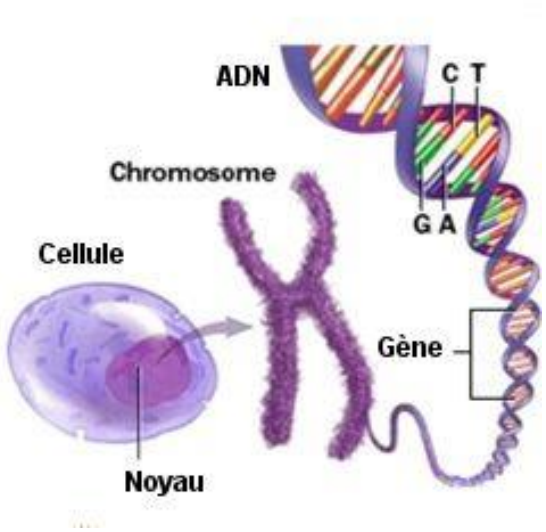


Figure 3 : Représentation de l'ADN

Au siècle dernier, Charles Darwin observa les phénomènes naturels et fit les constatations suivantes :

- L'évolution n'agit pas directement sur les êtres vivants, Elle opère en réalité sur les chromosomes contenus dans leur ADN ;
- L'évolution a deux composantes : la sélection et la reproduction ;

- La sélection garantit une reproduction plus fréquente des chromosomes les plus forts ;
- La reproduction est la phase durant laquelle s'effectue l'évolution

Dans les années 60s, John Holland expliqua comment ajouter de l'intelligence dans un programme informatique avec les croisements (échangeant le matériel génétique), la mutation (source de la diversité génétique) et la sélection (garantir la convergence).

Il formalisa ensuite les principes fondamentaux des algorithmes génétiques :

- La capacité de représentations élémentaires, comme les chaînes de bits, à coder des structures complexes ;
- Le pouvoir de transformations élémentaires à améliorer de telles structures.

Et récemment, David Goldberg ajouta à la théorie des algorithmes génétiques les idées suivantes :

- Un individu est lié à un environnement par son code d'ADN ;
- Une solution est liée à un problème par son indice de qualité.

2. Définition :

L'algorithme génétique (AG), fut développé par Holland, est un algorithme de recherche basé sur les mécanismes de la sélection naturelle et de la génétique. Il combine une stratégie de survie des plus forts avec un échange d'information aléatoire mais structuré. Pour un problème pour lequel une solution est inconnue, un ensemble des solutions possibles est créé aléatoirement. On appelle cet ensemble la population. Les caractéristiques (ou variables à déterminer) sont alors utilisées dans des séquences de gènes qui seront combinées avec 23 d'autres gènes pour former des chromosomes et par après des individus. Chaque solution est associée à un individu, et cet individu est évalué et classifié selon sa ressemblance avec la meilleure, mais encore inconnue, solution au problème. Il peut être démontré qu'en utilisant un processus de sélection naturelle inspiré de Darwin, cette méthode convergera graduellement à une solution. Comme dans les systèmes biologiques soumis à des contraintes, les meilleurs individus de la population sont ceux qui ont une meilleure chance de se reproduire et de transmettre une partie de leur héritage génétique à la prochaine génération. Une nouvelle population, ou génération, est alors créée en combinant les gènes des parents. On s'attend à ce que certains individus de la nouvelle génération possèdent les meilleures caractéristiques de leurs deux parents, et donc qu'ils seront meilleurs et

seront une meilleure solution au problème. Le nouveau groupe (la nouvelle génération) est alors soumis aux mêmes critères de sélection, et par après génère ses propres rejetons. Ce processus est répété plusieurs fois, jusqu'à ce que tous les individus possèdent le même héritage génétique. Les membres de cette dernière génération, qui sont habituellement très différents de leurs ancêtres, possèdent de l'information génétique qui correspond à la meilleure solution au problème. L'algorithme génétique de base comporte trois opérations simples qui ne sont pas plus compliquées que des opérations algébriques :

- Sélection
- Reproduction
- Mutation

3. Principe :

L'algorithme génétique est basé sur les étapes suivantes :

1. Création d'une population initiale
2. Evaluation des individus de la population
3. Sélection des " meilleurs" individus
4. Croisement et mutation dans le « mating pool »
5. Formation d'une nouvelle génération

Retour au 2 si non convergence

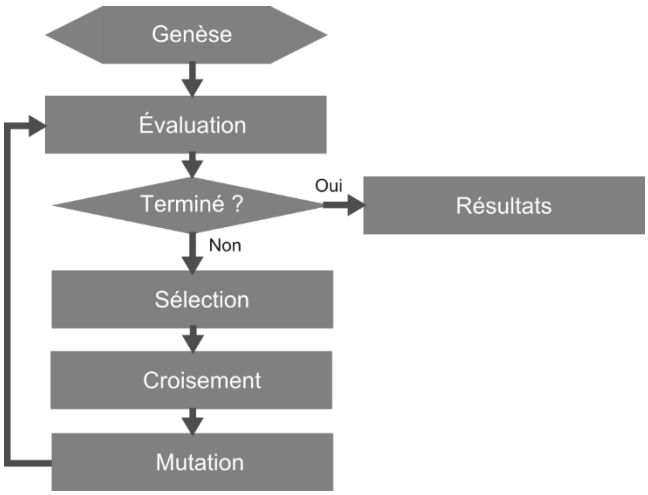


Figure 4: Organigramme d'algorithme génétique

II. Les opérateurs:

1. Sélection :

La sélection a pour objectif d'identifier les individus qui doivent se reproduire. Cet opérateur ne crée pas de nouveaux individus mais identifie les individus sur la base de leur fonction d'adaptation, les individus les mieux adaptés sont sélectionnés alors que les moins bien adaptés sont écartés. La sélection doit favoriser les meilleurs éléments selon le critère à optimiser (minimiser ou maximiser). Ceci permet de donner aux individus dont la valeur est plus grande une probabilité plus élevée de contribuer à la génération suivante.

Il existe plusieurs méthodes de sélection, les plus connues étant la roue de la fortune, la sélection par tournoi, élitisme, N/2-élitisme et sélection par rang.

Roue de la fortune (roulette) :

La « roue de la fortune » est la plus ancienne, où chaque individu, de la population de taille maximale j_{max}, occupe une section de la roue proportionnellement à sa fonction d'adaptation $Fitness(j)$, la probabilité de sélection d'un individu (j) s'écrit comme suit :

$$prob(j) = Fitness(j) / \sum Fitness(i)$$

À chaque fois qu'un individu doit être sélectionné, un tirage à la loterie s'effectue et propose un candidat, les individus possédant une plus grande fonction d'adaptation ayant plus de chance d'être sélectionnés.

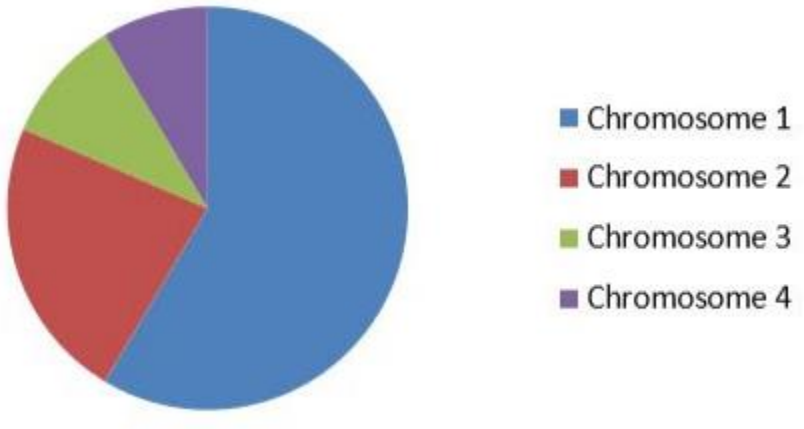


Figure 5 : Exemple de sélection par roue de la fortune

Sélection par tournoi :

A chaque fois qu'il faut sélectionner un individu, la « sélection par tournoi » consiste à tirer aléatoirement (k) individus de la population, sans tenir compte de la valeur de leur fonction d'adaptation, et de choisir le meilleur individu parmi les k individus. Le nombre d'individus sélectionnés a une influence sur la pression de sélection, lorsque k = 2, la sélection est dite par « tournoi binaire ».

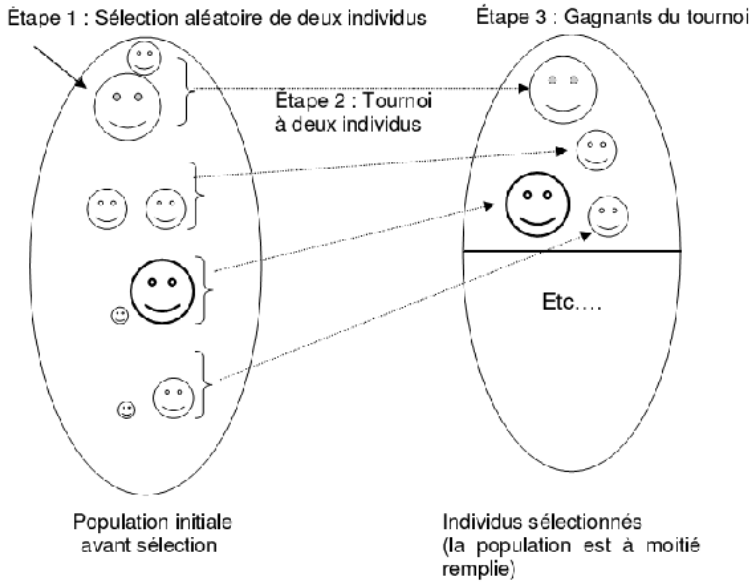


Figure 6 : Représentation d'une sélection par tournoi d'individus pour un critère de minimisation. Chaque individu représente une solution possible

Élitisme :

Une technique très souvent utilisée avec l'algorithme génétique est l'élitisme. L'élitisme est la conservation du meilleur individu dans la génération suivante. Après le croisement, on compare le meilleur individu de la nouvelle génération avec le meilleur individu de la génération précédente. Le meilleur individu est conservé et l'autre est détruit. Cette technique permet de s'assurer que la fonction objective augmente (diminue) toujours si on maximise (minimise) une fonction.

N/2 - Élitisme :

Les individus sont triés selon leur fonction d'adaptation, seule la moitié supérieure de la population correspondant aux meilleurs composants est sélectionnée, nous avons constaté que la pression de sélection est trop forte, il est important de maintenir une diversité de gènes pour les utiliser dans la population suivante et avoir des populations nouvelles quand on les combine.

Sélection par rang :

La sélection par roulette présente des inconvénients lorsque la valeur d'évaluation des individus varie énormément. En effet, on risquerait d'atteindre une situation de stagnation de l'évolution. Imaginons le cas ou 90% de la roulette est allouée à l'individu qui a la meilleure évaluation, alors les autres individus auront une probabilité très faible d'être sélectionnés. 28 La sélection par rang trie d'abord la population par évaluation. Ensuite, chaque individu se voit associé un rang en fonction de sa position.

Ainsi le plus mauvais individu aura le rang 1, le suivant 2, et ainsi de suite jusqu'au meilleur individu qui aura le rang N, pour une population de N individus.

La sélection par rang d'un individu est identique à la sélection par roulette, mais les proportions sont en relation avec le rang plutôt qu'avec la valeur de l'évaluation. Avec cette méthode de sélection, tous les individus ont une chance d'être sélectionnés. Cependant, elle conduit à une convergence plus lente vers la bonne solution. Ceci est dû au fait que les meilleurs individus ne diffèrent pas énormément des plus mauvais.

2. Croissement :

Le phénomène de croisement est une propriété naturelle de l'ADN, et c'est analogiquement qu'on fait les opérations de croisement dans les AG.

Lors de cette opération, deux chromosomes s'échangent des parties de leurs chaînes, pour donner de nouveaux chromosomes. Ces enjambements peuvent être simples ou multiples.

Sa probabilité d'apparition lors d'un croisement entre deux chromosomes est un paramètre de l'algorithme génétique et dépend du problème.

a. Croisement binaire :

Croisement en un point : on choisit au hasard un point de croisement, pour chaque couple. Notons que le croisement s'effectue directement au niveau binaire, et non pas au niveau des gènes. Un chromosome peut donc être coupé au milieu d'un gène.

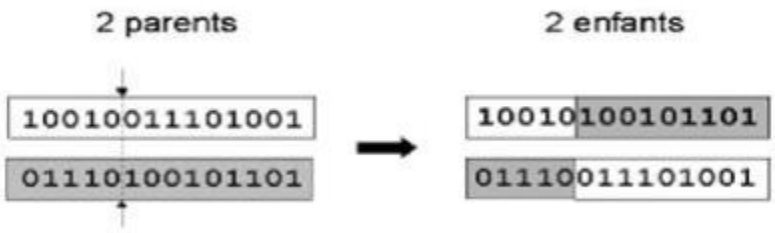


Figure 7 : Représentation schématique du croisement en 1 point

Croisement en deux points : On choisit au hasard deux points de croisement. Par la suite, nous avons utilisé cet opérateur car il est généralement considéré comme plus efficace que le précédent

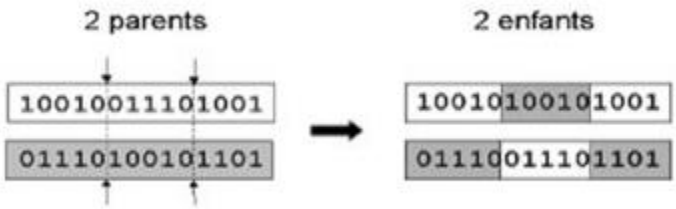


Figure 8 : Représentation schématique du croisement en 2 points

Notons que d'autres formes de croisement existent, du croisement en k points jusqu'au cas limite du croisement uniforme.

b. Croisement réel :

Le croisement réel ne se différencie du croisement binaire que par la nature des éléments qu'il altère : ce ne sont plus des bits qui sont échangés à droite du point de croisement, mais des variables réelles.

3. Mutation :

Nous définissons une mutation comme étant l'inversion d'un bit dans un chromosome. Cela revient à modifier aléatoirement la valeur d'un paramètre du dispositif. Les mutations jouent le rôle de bruit et empêchent l'évolution de se figer. Elles permettent d'assurer une recherche aussi bien globale que locale, selon le poids et le nombre des bits mutés. De plus, elles garantissent mathématiquement que l'optimum global peut être atteint.

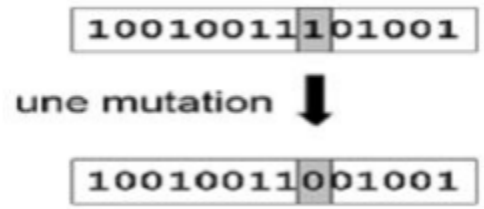


Figure 9 : Représentation schématique d'une mutation dans un chromosome

La fonction de mutation prend en paramètre le nombre de mutation et la probabilité de la mutation en supplémentaire.

III. Adaptation de l'algorithme génétique au KSP :

Nous allons maintenant adapter l'algorithme génétique à la résolution du problème de sac à dos. Le but sera ici de montrer comment modéliser le problème à partir de l'algorithme génétique et les divers opérateurs que nous avons à disposition, qui ont été définis antérieurement.

Individu : La solution possible. (Les objets à mettre dans le sac)
Population : un ensemble de solutions

Sélection naturelle :
Qui sélectionne les meilleurs individus de la population (les combinaisons réalisables avec le maximum de valeurs) et élimine les individus les moins adaptés.

Croisement :
Etant donné deux combinaisons, il faut les combiner pour en construire deux

autres.
Nous pouvons suivre la méthode suivante :
1. On choisit aléatoirement deux génomes proportionnellement avec leur valeur.
2. On choisit un indice aléatoirement entre 0 et la longueur du génome.
3. On couple les deux génomes dans cet indice, cela nous donne quatre parties.
4. On relie la première partie du génome 1 avec la deuxième partie du génome 2, et vice versa.

Mutation :
Il s'agit ici de modifier un ou plusieurs bits avec une tel probabilité.

Chapitre 3 : Implémentation

I. L'environnement de travail :

Python :

Python est un langage de programmation interprété, multi paradigme et multiplateformes. Il favorise la programmation impérative structurée, fonctionnelle et orientée objet. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions ; il est ainsi similaire à Perl, Ruby, Scheme, Smalltalk …

Bibliothèque :

On va également utiliser les bibliothèques suivantes : random, typing et collection

```
import random
from typing import List, Tuple
from collections import namedtuple
import time
```

II. Le code :

1. Le génome et la population :

Au début en doit générer un génome, un génome dans notre cas est une liste des zéros et des uns, où un réfère à un indice d'un objet inclue dans le sac et le zéro, l'indice d'un objet pas inclue dans le sac.

```
Genome = List[int]
```

Python est un langage à typage fort dynamique (pas besoin de déclarer les type), mais je vais le typer pour améliorer la lisibilité de mon code.

```
def generate_genome(lenght: int) -> Genome  :
    return random.choices([0,1],k=lenght)
```

La fonction random.choices retourne une liste de longueur K (lenght) des éléments choisie de la population [0,1].

Ce qui concerne la population, c'est une liste des génomes :

```
Population = List[Genome]
def generate_population(size :int,lenght : int) -> Population :
    return [generate_genome(lenght) for _ in range(size)]
```

Où size : Taille de la population (nombre de génome)

Lenght : longueur du génome.

2. Fonction fitness :

La fonction fitness calcule la valeur d'un génome, calcule la somme des valeurs des objets inclue dans le sac en respectant le poids maximal du sac.

```
def fitness(genome :Genome,things :[Thing],Wmax :int):
    if len(genome) != len(things) :
        raise ValueError("things and genome not the same lenght")

    value = weight = 0
    for i , thing in enumerate(things) :
        if genome[i] == 1 :
            value += thing.value
            weight += thing.weight
```

```
        if weight > Wmax :
            return 0
    return value
```

Où Wmax : Le poids maximal du sac

things : L ‘ensemble des objets

```
Thing = namedtuple('Thing',['name','value','weight'])
things = [
    Thing('thing1',500,2200),
    Thing('thing2',150,160),
    Thing('thing3',60,350),
    Thing('thing4',40,333),
    Thing('thing5',30,192),
    Thing('thing5',15,50),
    Thing('thing5',300,200),
    Thing('thing5',90,19),
    Thing('thing5',33,100),
]
```

un objet est caractérisé par un nom, valeur et poids

3. Fonction de sélection :

Cette fonction va choisir aléatoirement deux génomes en fonction de leur valeur.

```
def selection_pair(population :Population,fitness) -> Tuple[Genome,Genome]:
    return random.choices(population=population,weights=[fitness(genome) for genome in population],k=2)
```

Dans cette fonction, on a ajouté la fonction fitness dans les arguments pour que cette méthode soit général dans le cas d’un fitness différente, et on remarque qu’on a donné un argument à la fonction fitness et normalement ça prend trois, cela reste valide car on va utiliser la fonction PARTIAL ultérieurement.

4. Fonction de croissement :

```python
def single_point_crossover(genome1 :Genome,genome2 :Genome) -> Tuple[Genome,Genome]:
    if len(genome1) != len(genome2) :
        raise ValueError("G1 and G2 not the same lenght")

    if len(genome1) < 2:
        return genome2 ,genome1

    p = random.randint(1,len(genome1)-1)
    return genome1[0:p] + genome2[p:] , genome2[0:p] + genome1[p:]
```

Ceci est une implémentation directe de ce qu'il est déjà expliqué.

5. Fonction de mutation :

```python
def mutation(genome : Genome,number_of_mutation :int=1, prob : float=0.5) -> Genome:
    for _ in range(number_of_mutation):
        index = random.randrange(len(genome))
        genome[index] = genome[index] if random.random() > prob else abs(genome[index]-1)
    return genome
```

Ceci est une implémentation directe de ce qu'il est déjà expliqué.

6. Fonction d'évolution :

Pour la fonction d'évolution :

```python
def run_evolution(fitness_funct, Generat_pop_funct, fitness_limit :int,generation_limit :int=100):
```

Cette fonction termine quand on atteint la valeur limite (fitness_limit) ou bien la génération limite.

La première étape est de générer une population :

```python
    P = Generat_pop_funct()
```

Puis on va itérer les générations :

```
    for i in range(generation_limit):
```

```
        P = sorted(P, key=lambda genome: fitness_funct(genome),reverse=True )

        if fitness_funct(P[0]>= fitness_limit):
            break
        next_generation = P[0:2]
```

Pour chaque génération, on trie la population en fonction des valeurs, puis on passe les deux meilleurs individues directement à la génération suivante, avec un petit test si la valeur limite est atteinte.

Maintenant on a que deux nouveaux individue dans la nouvelle génération, on doit trouver les autres individues comme suivant :

```
for j in range( int(len(P)/2 )-1):
            parents = selection_pair(P,fitness_funct)
            g1,g2 = single_point_crossover(parents[0],parents[1])
            g1 = mutation(g1)
            g2 = mutation(g2)
            next_generation += [g1,g2]
```

Cette boucle est incluse dans la première.

Chaque itération de cette boucle produit deux nouveaux génomes, alors il faut itérer que la moitié de la population, et on soustrait un, car on a déjà choisi deux génomes dans la phase d'avant.

Dans cette phase on a utilisé la fonction sélection, croissement et mutation.

A la fin de chaque itération on ajoute les deux nouveaux génomes à la nouvelle génération.

Après qu'on a générer une nouvelle génération on l'affecte à la variable P(population) dehors de la deuxième boucle.

```
P = next_generation
```

Pour la valeur à retourner j'ai choisi de retourner le meilleur génome avec le numéro de la génération

```
    return P[0],i
```

Cela est valable si on sort de la fonction à cause de fitness_limit , alors pour traiter le cas de generation_limit, on va ajouter cette ligne avant la fin :

```
    P = sorted(P, key=lambda genome: fitness_funct(genome),reverse=True )
```

La function complete:

```
def run_evolution(fitness_funct, Generat_pop_funct, fitness_limit :int,generation_limit :int=100):

    P = Generat_pop_funct()

    for i in range(generation_limit):
        P = sorted(P, key=lambda genome: fitness_funct(genome),reverse=True )
        #if 0 not in P[0] :
        #    break
        if fitness_funct(P[0])>= fitness_limit):
            break
        next_generation = P[0:2]

        for j in range( int(len(P)/2 )-1):
            parents = selection_pair(P,fitness_funct)
            g1,g2 = single_point_crossover(parents[0],parents[1])
            g1 = mutation(g1)
            g2 = mutation(g2)
            next_generation += [g1,g2]

        P = next_generation

    P = sorted(P, key=lambda genome: fitness_funct(genome),reverse=True )

    return P[0],i
```

7. Test :

Avant de commencer l'évolution, je vais faire une petite explication de la fonction PARTIAL :

Les fonctions partielles permettent de dériver une fonction avec des paramètres x en une fonction avec moins de paramètres et des valeurs fixes définies pour la fonction la plus limitée.

Exemple :

On déclare les objets :

```
things = [
    Thing('thing1',500,2200),
    Thing('thing2',150,160),
    Thing('thing3',60,350),
    Thing('thing4',40,333),
    Thing('thing5',30,192),
    Thing('thing5',15,50),
    Thing('thing5',300,200),
    Thing('thing5',90,19),
    Thing('thing5',33,100),
]
```

Puisque c'est difficile d'analyser le résultat car je ne connais pas la solution de ce exemple, je vais eliminier le test de fitness_limit, et je vais laisser generation_limit pour voir l'evolution du résultat :

Pour cela je vais ajouter cette ligne dans la premiere boucle :

```
print("value :",fitness_funct(P[0]),"genome :",P[0],"Generation :",i)
```

Le test est :

```
Wmax=1000   #max weight
sizeP=10    #size of population
genome,generation = run_evolution( fitness_funct=partial(fitness,things=things,Wmax=Wmax),
                                   Generat_pop_funct=partial(generate_population,size=sizeP,lenght=len(things)),
                                   generation_limit=100
                                 )
```

Pour un poids max de 1000, une population de 10 individues, et une génération limite de 100, on a le résultat suivant :

```
value : 328 ||  genome : [0, 1, 0, 1, 0, 1, 0, 1, 1] ||  Generation : 0
value : 343 ||  genome : [0, 1, 0, 1, 1, 0, 0, 1, 1] ||  Generation : 1
value : 355 ||  genome : [0, 1, 1, 1, 0, 1, 0, 1, 0] ||  Generation : 2
value : 355 ||  genome : [0, 1, 1, 1, 0, 1, 0, 1, 0] ||  Generation : 3
value : 355 ||  genome : [0, 1, 1, 1, 0, 1, 0, 1, 0] ||  Generation : 4
value : 355 ||  genome : [0, 1, 1, 1, 0, 1, 0, 1, 0] ||  Generation : 5
value : 373 ||  genome : [0, 1, 1, 1, 0, 0, 0, 1, 1] ||  Generation : 6
value : 573 ||  genome : [0, 1, 0, 0, 0, 0, 1, 1, 1] ||  Generation : 7
value : 613 ||  genome : [0, 1, 0, 1, 0, 0, 1, 1, 1] ||  Generation : 8
value : 613 ||  genome : [0, 1, 0, 1, 0, 0, 1, 1, 1] ||  Generation : 9
value : 613 ||  genome : [0, 1, 0, 1, 0, 0, 1, 1, 1] ||  Generation : 10
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 11
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 12
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 13
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 14
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 15
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 16
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 17
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 18
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 19
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 20
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 21
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 22
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 23
```

Le résultat continue jusqu'à génération : 100 (génération_limit)

Et le dernier changement et dans la génération 38 :

```
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 36
value : 628 ||  genome : [0, 1, 0, 1, 0, 1, 1, 1, 1] ||  Generation : 37
value : 648 ||  genome : [0, 1, 1, 0, 0, 1, 1, 1, 1] ||  Generation : 38
value : 648 ||  genome : [0, 1, 1, 0, 0, 1, 1, 1, 1] ||  Generation : 39
```

8. Test général :

Dans cette partie on va exécuter un test général, pour tester le code dans des exemples avec un grand nombre d'objets et on va calculer le temps d'exécution, et le poids maximal du sac va être infinie, ça veut dire que le meilleur résultat c'est un génome qui contient que des uns.

Alors notre fonction d'évolution va s'arrêter si elle atteinte le nombre de génération maximal ou bien elle atteint la solution [1,1, 1, ...,1] :

```
if 0 not in P[0] :
            break
```

On commence avec une fonction qui générer les objets :

```
def generate_thing(num_thing):
    things = []
    for _ in range(num_thing):
        things.append( Thing('',random.randrange(1,100),random.randrange(1,100)) )
    return things
```

La valeur et le poids sont entre 1 et 99.

```
print("item  | generation | time | rate | genome")

for i in range(1,40):
    X= generate_thing(i)


    Wmax=i*100   #max weight
    sizeP=10    #size of population

    start = time.time()
    genome,generation = run_evolution( fitness_funct=partial(fitness,things=X,Wmax=Wmax),
                                       Generat_pop_funct=partial(generate_population,size=sizeP,lenght=len(X)),
                                       generation_limit=100
                                     )
    end = time.time()

    s=0
    for h in genome : #for the calcule of the rate
        if h == 0 :
            s+=1

    print(i," | ",generation," | ",format(end-start,".2e")," | ",round((i-s)/i*100,2),"% | ",genome)
```

Résultat :

```
item  | generation | time | rate | genome
1  |  0  |  0.00e+00  |  100.0 % |  [1]
2  |  0  |  0.00e+00  |  100.0 % |  [1, 1]
3  |  0  |  0.00e+00  |  100.0 % |  [1, 1, 1]
4  |  3  |  9.98e-04  |  100.0 % |  [1, 1, 1, 1]
5  |  1  |  9.97e-04  |  100.0 % |  [1, 1, 1, 1, 1]
6  |  2  |  0.00e+00  |  100.0 % |  [1, 1, 1, 1, 1, 1]
7  |  5  |  1.99e-03  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1]
8  |  3  |  9.97e-04  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1]
9  |  9  |  2.00e-03  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1]
10  |  6  |  1.99e-03  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
11  |  4  |  1.00e-03  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
12  |  30  |  6.98e-03  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
13  |  15  |  4.98e-03  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
14  |  25  |  6.99e-03  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
15  |  23  |  7.98e-03  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
16  |  16  |  6.98e-03  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
17  |  35  |  1.20e-02  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

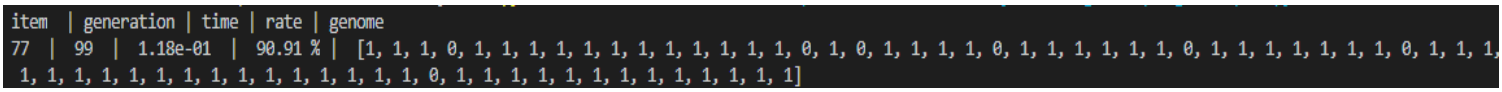
On remarque que quelque cas, le nombre de génération égal à 0, car la solution est directement générée

```
35  |  88  |  5.88e-02  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
36  |  99  |  6.48e-02  |  97.22 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1]
37  |  99  |  5.09e-02  |  97.3 % |  [1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
38  |  71  |  4.19e-02  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
39  |  62  |  4.29e-02  |  100.0 % |  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Et il y’a des cas où ils ont atteint la génération 100 sans trouver la solution, mais une solution accès proche de 97%, et si on augmente la population, ces tests vont trouver la solution plus rapidement (moins de génération).

9. Comparaison :

On a trouvé auparavant qu’avec la méthode naïve, un problème de 77 objets prend $154\ 153\ 413$ années pour trouver la solution exacte. Avec notre algorithme :



Le résultat est proche de 90% de la solution optimale dans 0.118 secondes.

Un résultat qu’il est très satisfaisant par rapport à 154 153 413 années.

Conclusion

J’ai présenté la modélisation du problème du sac à dos au modelé génétique ,Une simple implémentation a pu réduire le temps d’exécution de 154 153 413 années à 0.118 secondes avec une précision de 90.91% ce que nous permettre de dire que cette approche est pratiquement réussi dans tel NP-complet problème.

Références

Partie théorique:

1. AG : https://fr.wikipedia.org/wiki/Algorithme_g%C3%A9n%C3%A9tique
2. L'Algorithme génétique : http://www.chez.com/produ/badro/
3. RAPPORT DU PROJET DE FIN D'ANNÉE Réalisation d'Une Application Pour La Résolution Du Problème d'Optimisation Du Voyageur De Commerce, Réalisé Par: Encadré Par: ● BOULLAFT Rafik ● ZAKRITI Said ● EL BOUSSALMANI Oualid , Soutenu le : 20 juin 2020 , ENSAH

Partie code :

1. AG Code : https://www.youtube.com/watch?v=nhT56blfRpE&t=632s