Algorithme génétique : Problème du sac à dos
Algorithmes de recherche

Jean-Marie SAINDON

15 janvier 2020

1 Implémentation

La résolution du problème du sac à dos par le biais d'un algorithme génétique a été implémenté en python de la façon suivante :

Tout d'abord, 4 classes principales ont été créées : Parameters, Object, Individual et Population. Parameters est une classe regroupant simplement les différentes variables essentielles du problème du sac à dos et de la résolution génétique, à savoir : les valeurs maximales et minimales des coût et poids des objets, les objets eux-mêmes, la taille des populations, le nombre de génération déroulées par les simulation, etc. Object est, quant à elle, une classe simple représentant un objet, avec donc son coût et son poids.

```python
class Parameters:
    Seed = 9876
    MaxFill = 60
    NumberOfPopulations = 100
    NumberOfGenerations = 100
    PopulationSize = 50
    MutationRate = 5

    # Objects configuration
    NumberOfObjects = 40
    InfCostObjects = 20
    MaxCostObjects = 300
    InfWeightObjects = 1
    MaxWeightObjects = 6
    Objects = None
```

```python
class Object:
    def __init__(self, id, cost, weight):
        self._id = id
        self._cost = cost
        self._weight = weight
```

FIGURE 1 – Extrait des classes Parameters et Object

Individual est une classe équivalente à ce que l'on pourrait appeler chromosome dans d'autres implémentations d'algorithmes génétiques. La classe opte pour une représentation de solution au problème sous forme de tableau numpy remplit de 0 ou de 1 selon la présence ou non de l'objet correspondant dans la liste des objets. Enfin, Population contient simplement une liste de Individual.

```python
class Individual:
    _param = None
    _pack = None
    _sumObj = 0
    _sumWeight = 0
    _sumCost = 0

    def __init__(self, param):
        self._param = param

    def setPack(self, pack):
```

```python
class Population:
    _members = None
    _param = None

    def __init__(self, param):
        self._param = param

    def fillRandomly(self):
        self._members = [Individual(self._param).fillRandomly() for k in range(self._param.PopulationSize)]
        return self

    def reproduce(self, p1, p2):
```

FIGURE 2 – Extrait des classes Individual et Population

2 Déroulement du procédé et résultats

Afin de dérouler l'algorithme génétique et de trouver les meilleurs paramètres pour aboutir à la solution optimale, on génère une centaine de populations différentes avec des tailles et des taux de mutation aléatoires. Chacune de ces populations évolue sur 100 générations et le meilleur individu de la dernière évolution est sélectionné pour représenter sa population. Le meilleur représentant (celui ayant la meilleur maximisation du coût) est alors choisi comme solution finale de l'algorithme.

Mon implémentation offre des résultats satisfaisants en suivant ce procédé. Sur quelques simulations, le pourcentage moyen d'optimalité tournais autour des 90 % (avec notamment 91.37 % sur les simulations représentées ci-dessous).

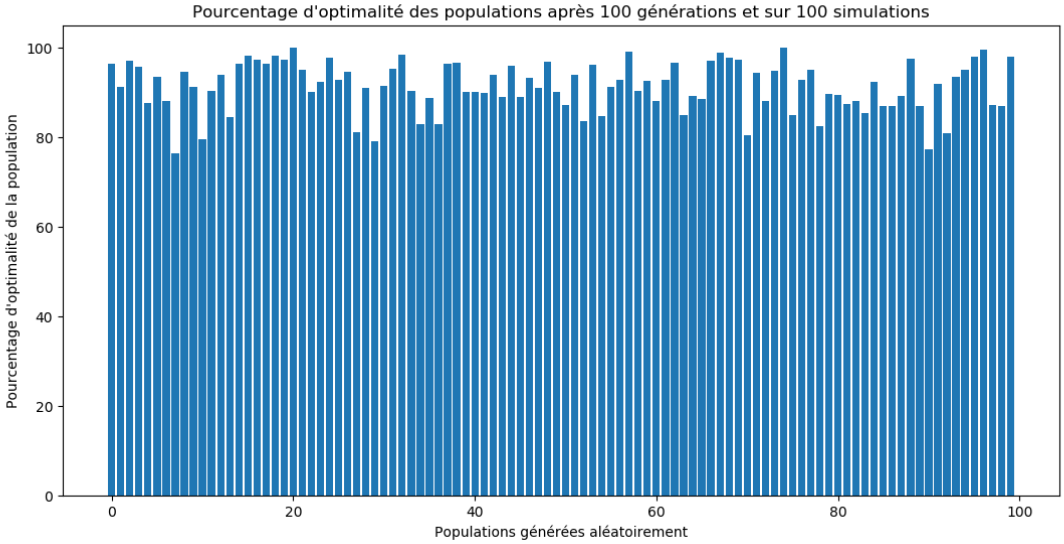

FIGURE 3 – Graphique représentant le pourcentage d'optimalité des 100 populations générées

La vérification de l'optimalité se fait par rapport au résultat de la librairie knapsack de python. (Attention cette librairie est extrêmement lente)

```
-X-XX-XXX---XX-XX-XXXXXXXX-X--X--XXXXXXX (Size = 27) (Weight = 60) (Cost = 4314) (BestCost = 4329) 97
-X-XXXXXXXX-X--XX--XXX-XXX-XXXX-XXXXXXX- (Size = 29) (Weight = 60) (Cost = 3771) (BestCost = 4329) 98
--XXXXXXX---X--XX-XXX--XXXXXX-X-XX-X-XXX (Size = 26) (Weight = 59) (Cost = 3764) (BestCost = 4329) 99
-X-XXXXXX---XX-XX-XXXX-XXX-XXXX--XXXXXXX (Size = 29) (Weight = 60) (Cost = 4239) (BestCost = 4329) 100

Best solution :
-X-XX-XXX---XX-XX-XXXXXXXX-XXX---XXXXXXX (Size = 28) (Weight = 60) (Cost = 4329)
Seed : 70978 , Population size : 532 , Mutation rate : 10

Solution of the knapsack module :
4329

Optimality percentage : 91.37167937167936
```

FIGURE 4 – Output du script