Exercício

José Lucas Damasceno Holanda

1. Resolver o problema da mochila como um PPI.

```
"""
Created on Tue Sep 21 16:26:42 2021

@author: José Lucas Damasceno
"""


import cvxpy as cp

# Create two scalar optimization variables.
x = cp.Variable(5)
p = [2, 1, 4, 5, 3]
l = [79, 17, 187, 245, 140]


# Create two constraints.
constraints = [x[:] >= 0,
               x[:] <= 10,
               x[1]*p[1] + x[2]*p[2] + x[3]*p[3] +
               x[4]*p[4] + x[0]*p[0] <= 10 ]

# Form objective.
obj = cp.Maximize(x[1]*l[1] + x[2]*l[2] + x[3]*l[3] +
                  x[4]*l[4] + x[0]*l[0])

# Form and solve problem.
prob = cp.Problem(obj, constraints)
prob.solve()  # Returns the optimal value.
print("Status: ", prob.status)
print("Valor otimizado: ", round(prob.value))

x = x.value

print("Quantidades por produto: \n",
                                  round(x[0]),
                              "\n",round(x[1]),
                              "\n",round(x[2]),
                              "\n",round(x[3]),
                              "\n",round(x[4]))
```

Saída na janela de comando:

```
Status:  optimal
Valor otimizado:  490
Quantidades por produto:
 0
 0
 0
 2
 0
```

---

2. Aplicar um algoritmo genético para resolver o problema da mochila. Empregar o exemplo dado aqui apenas para validar o código. Isto é, o código deve ser geral, para quaisquer capacidade da mochila, número de artigos, pesos e expectativas de lucro.

```
"""
Created on Mon Sep 27 09:53:24 2021

@author: José Lucas Damasceno
"""

import numpy as np
import random as rd
from random import randint


weight = [2, 1, 4, 5, 3]
value  = [79, 17, 187, 245, 140]
knapsack_threshold = 10    #Maximum weight that the bag of thief can hold

numberMax = 3; # Quantidade máxima de itens individualmente

item_number = np.arange(1,len(weight)+1)

print('Segue a lista implementada:')
print('Item No.   Peso   Valor')
for i in range(item_number.shape[0]):
    print('{0}          {1}         {2}\n'.format(item_number[i], weight[i],
value[i]))


solutions_per_pop = 100
pop_size = (solutions_per_pop, item_number.shape[0])
print('Tamanho da População = {}'.format(pop_size))
initial_population = np.random.randint(numberMax, size = pop_size)
initial_population = initial_population.astype(int)
num_generations = 10000
print('População inicial: \n{}'.format(initial_population))

def cal_fitness(weight, value, population, threshold):
    fitness = np.empty(population.shape[0])
```

```
    for i in range(population.shape[0]):
        S1 = np.sum(population[i] * value)
        S2 = np.sum(population[i] * weight)
        if S2 <= threshold:
            fitness[i] = S1
        else :
            fitness[i] = 0
    return fitness.astype(int)

def selection(fitness, num_parents, population):
    fitness = list(fitness)
    parents = np.empty((num_parents, population.shape[1]))
    for i in range(num_parents):
        max_fitness_idx = np.where(fitness == np.max(fitness))
        parents[i,:] = population[max_fitness_idx[0][0], :]
        fitness[max_fitness_idx[0][0]] = -999999
    return parents

def crossover(parents, num_offsprings):
    offsprings = np.empty((num_offsprings, parents.shape[1]))
    crossover_point = int(parents.shape[1]/2)
    crossover_rate = 0.8
    i=0
    while (parents.shape[0] < num_offsprings):
        parent1_index = i%parents.shape[0]
        parent2_index = (i+1)%parents.shape[0]
        x = rd.random()
        if x > crossover_rate:
            continue
        parent1_index = i%parents.shape[0]
        parent2_index = (i+1)%parents.shape[0]
        offsprings[i,0:crossover_point] = parents[parent1_index,0:crossover_point]
        offsprings[i,crossover_point:] = parents[parent2_index,crossover_point:]
        i=+1
    return offsprings

def mutation(offsprings):
    mutants = np.empty((offsprings.shape))
    mutation_rate = 0.4
    for i in range(mutants.shape[0]):
        random_value = rd.random()
        mutants[i,:] = offsprings[i,:]
        if random_value > mutation_rate:
            continue
        int_random_value = randint(0,offsprings.shape[1]-1)
        if mutants[i,int_random_value] == 0 :
            mutants[i,int_random_value] = 1
        else :
            mutants[i,int_random_value] = 0
    return mutants

def optimize(weight, value, population, pop_size, num_generations, threshold):
    parameters, fitness_history = [], []
    num_parents = int(pop_size[0]/2)
```

```
    num_offsprings = pop_size[0] - num_parents
    for i in range(num_generations):
        fitness = cal_fitness(weight, value, population, threshold)
        fitness_history.append(fitness)
        parents = selection(fitness, num_parents, population)
        offsprings = crossover(parents, num_offsprings)
        mutants = mutation(offsprings)
        population[0:parents.shape[0], :] = parents
        population[parents.shape[0]:, :] = mutants

    print('Última Geração: \n{}\n'.format(population))
    fitness_last_gen = cal_fitness(weight, value, population, threshold)
    print('Fitness da última geração: \n{}\n'.format(fitness_last_gen))
    max_fitness = np.where(fitness_last_gen == np.max(fitness_last_gen))
    parameters.append(population[max_fitness[0][0],:])
    return parameters, fitness_history


parameters, fitness_history = optimize(weight, value, initial_population,
pop_size, num_generations, knapsack_threshold)
print('Quantidade otimizada de itens: \n{}'.format(parameters))
```

A saída do código genético para este problema foi:

```
Segue a lista implementada:
Item No.   Peso   Valor
1          2         79

2          1         17

3          4         187

4          5         245

5          3         140

Tamanho da População = (100, 5)
População inicial:
[[1 2 1 0 1]
 [2 2 2 1 2]
 ...
 [2 1 2 2 0]
 [1 1 0 1 0]]
Última Geração:
[[0 0 0 2 0]
 [0 0 0 2 0]
 ...
 [0 0 1 2 0]
 [0 0 0 2 0]]

Fitness da última geração:
[490 490 490 490 490 490 490 490 490 490 490 490 490 490 490 490 490 490
```

```
 490 490 490 490 490 490 490 490 490 490 490 490 490 490 490 490 490 490
 490 490 490 490 490 490 490 490 490 490 490 490 490 490   0 490   0 490
 490   0 490   0   0   0 490 490 490   0   0   0 490 490   0 490 490 490
   0   0   0   0   0 490   0   0   0 490   0   0 490   0   0 490 490 490
   0 490 490   0 490 490 490 490   0 490]

Quantidade otimizada de itens:
[array([0, 0, 0, 2, 0])]
```

---

3. Experimente o código da Figura 3, mediante alguns estudos de caso. Para isso faça o seguinte:

```python
def solve_knapsack_unbounded_bottomup(profits, weights, capacity):
    n = len(profits)
    # base checks
    if capacity <= 0 or n == 0 or len(weights) != n:
        return 0

    dp = [[-1 for _ in range(capacity + 1)] for _ in range(n)]
    # populate the capacity=0 columns
    for i in range(n):
        dp[i][0] = 0
    # process all sub-arrays for all capacities
    for i in range(n):
        for c in range(1, capacity + 1):
            profit1, profit2 = 0, 0
            if weights[i] <= c:
                profit1 = profits[i] + dp[i][c - weights[i]]
            if i > 0:
                profit2 = dp[i - 1][c]
                dp[i][c] = max(profit1, profit2)
                # maximum profit will be in the bottom-right corner.
                print_selected_items(dp, weights, capacity)
    return dp[n - 1][capacity]

def print_selected_items(dp, weights, capacity):
    idxes_list = []
    print("Selected weights are:", end=" ")
    n = len(weights)
    i = n - 1
    while i >= 0 and capacity >= 0:
        if i > 0 and dp[i][capacity] != dp[i - 1][capacity]:
            # include this item
            idxes_list.append(i)
            capacity -= weights[i]
        elif i == 0 and capacity >= weights[i]:
            # include this item
            idxes_list.append(i)
            capacity -= weights[i]
        else:
```

```
            i -= 1
    weights = [weights[idx] for idx in idxes_list]
    print(weights)
    print("Maximum Profit is:", end=" ")
    return weights

# Driver
profits = [540, 79, 17, 187, 245, 140]
weights = [5, 2, 1, 4, 5, 3]
capacity = 10
maximumProfit = solve_knapsack_unbounded_bottomup(profits, weights,
capacity)
print(maximumProfit)
```

a) Varie a capacidade da mochila, por exemplo, para 11kg e para 9kg.

Realizando esta alteração, obtém-se:

```
Selected weights are: [3, 3, 5]
Maximum Profit is: 525
```

b) Varie o lucro de algum artigo.

Realizando a alteração de valor do artigo com peso 1 para 170:

```
profits = [79, 170, 187, 245, 140]
weights = [2, 1, 4, 5, 3]
capacity = 11
```

Temos o seguinte resultado:

```
Selected weights are: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Maximum Profit is: 1870
```

c) Varie o peso de algum artigo.

Variando o peso do artigo 5 para 3:

```
profits = [79, 17, 187, 245, 140]
weights = [2, 1, 4, 3, 3]
capacity = 10
```

Constata-se que não é possível distinguir qual dos dois artigos 3 está sendo de fato considerado no problema de otimização:

```
Selected weights are: [3, 3, 3, 1]
Maximum Profit is: 752
```

d) Aumente o número de artigos.

Inserindo um artigo ao final do array com peso 6 e valor 500:

```
profits = [79, 17, 187, 245, 140, 500]
weights = [2, 1, 4, 5, 3, 6]
capacity = 10
```

Desta forma, a nova combinação de itens é:

```
Selected weights are: [6, 4]
Maximum Profit is: 687
```

e) Avalie a consistência do código a partir dos resultados encontrados.

O algoritmo apresenta uma ambiguidade de resoluções quando se insere artigos com mesmo peso.

---

4. Como os artigos selecionados são identificados por seus pesos, como funcionará o código da Tabela 3, caso haja artigos distintos (de lucros distintos) com mesmo peso? Para ter resposta a essa questão faça os seguintes experimentos:

a) Inclua no fim da lista de artigos um outro de peso 5 e valor 450.

Realizando esta alteração é possível obter um novo valor máximo para o problema:

```
profits  = [79, 17, 187, 245, 140, 540]
weights  = [2, 1, 4, 5, 3, 5]
capacity = 10
```

```
Selected weights are: [5, 5]
Maximum Profit is   : 1080
```

b) Reposicione o artigo F no início da lista.

Feito isso:

```
profits  = [540, 79, 17, 187, 245, 140]
weights  = [5, 2, 1, 4, 5, 3]
capacity = 10
```

```
Selected weights are: [5, 5]
Maximum Profit is   : 490
```

c) Procure explicar eventuais diferenças nos resultados dos itens acima.

Neste caso, o algoritmo não compara o valor máximo obtido com o próximo para se chegar a um resultado global. Apenas sobrescreve o resultado, obtendo-se, na saída, o último valor máximo obtido.

5. A partir da experiência com a programação dinâmica, qual sua avaliação dos algoritmos genéticos para solução do problema irrestrito da mochila? O que pode ser aproveitado do que foi feito na solução dos exercícios de 2 a 4 para melhorar o desempenho do algoritmo genético?

Devido à aleatoriedade da obtenção das populações no algoritmo genético, não necessariamente será obtido o valor global de otimização do problema da mochila. Para este problema o algoritmo genético não é tão eficiente com relação à precisão no valor global otimizado.

---

jose.holanda@ee.ufcg.edu.br
