Money Bills Problem

Task:

Utilizați un algoritm genetic pentru a rezolva problema: pentru plata unei sume S se pot folosi bancnote cu n valori nominale distincte (v[i], i=1,…,n). NU este disponibila bancnota cu valoare 1. Pentru fiecare valoare nominala este disponibila o cantitate limitata de bancnote (c[i], i=1,…,n). Găsiți o modalitate de plata care folosește numărul minim de bancnote. Este utilizat un algoritm genetic.

Individual representation:

We consider x an individual. For any x we have $x = \{x_0, x_1, \dots x_{n-1}\}$ where x_i or $x[i]$ represents the amount of bills of type $v[i]$ is used to pay the sum S. We cannot pay less than 0 banknotes, thus we have the constraint $0 \leq x[i] \leq c[i]$.

Example:

$for\ n = 7$

$v = [\ 2, \quad 3\ , \quad 5, \quad 7, \quad 10, \quad 11, \quad 13]$

$c = [\ 8, \quad 11, \quad 2, \quad 3, \quad 5\ , \quad 7\ , \quad 4]$

$x = [\ 2, \quad 7\ , \quad 1, \quad 0, \quad 2\ , \quad 0\ , \quad 0]$

→ we pay the sum S using 2 banknotes of value 2, 7 of value 3, 1 of value 5, none of value 7, 2 of value 10, none of value 11 and none of value 13.

Fitness function (lines 39 – 49):

Our goal is to pay the sum S using the minimum amount of bills. If we consider $cost = sum(x[i])$, $i = \overline{0, n-1}$ and we wish to minimize this cost, then our fitness function can be defined as such:

$$\boldsymbol{fitness_{max} = \frac{1}{1 + cost}}$$

We consider $1 + cost$ instead of just $cost$ to avoid division by zero in the case of $cost = 0$.

Restrictions / feasibility check (lines 53 – 59):

We must pay the whole sum S, which means that the value of $sum(x[i] \cdot v[i]), i = \overline{0, n-1}$ must be equal to S. If the condition is not met, the individual is non-feasible. Otherwise, the individual is feasible.

Initial population generation (lines 78 – 87):

Each individual x in the population is randomly generated, by assigning $x[j]$ a random value between 0 and $c[j]$, value that represents the chosen amount of banknotes of type $v[j]$.

```
while wasAccepted == False:
    # we generate element by element in the individual
    x_fitness = 0
```

```
    for j in range(parameters.n):
        current_value = np.random.randint(0, parameters.max_bill_amounts[j] + 1)
        x += [current_value

    # after we have an individual, we check if it is feasible
    if isFeasible(x) == True:
        wasAccepted = True
        pop += [x]  # we add x to the population
        x_fitness = fitness_func(x)
        init_pop_fitness += [x_fitness]
    else:
        x = []
```

After the generation, we check if the individual is feasible. If not, we try again. When we find a feasible individual, we add it to the initial population as well as its fitness value to our list of initial fitness values.

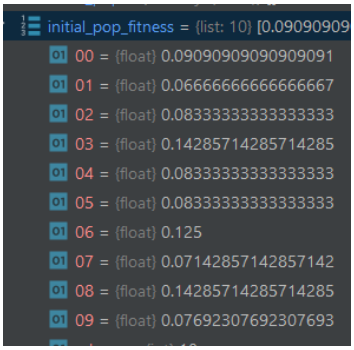
This mechanism will be repeated n times where $n = population\ size$.

Initial population example:

$\boldsymbol{for\ n = 7}$
$\boldsymbol{v = [2,\ 3,\ 5,\ 7,\ 10,\ 11,\ 13]}$
$\boldsymbol{c = [8,\ 11,\ 2,\ 3,\ 5,\ 7,\ 4]}$
$\boldsymbol{x = [2,\ 7,\ 1,\ 0,\ 2,\ 0,\ 0]}$
$\boldsymbol{population\ size = 10}$

initial_pop

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



Parent selection (lines 94 – 133):

Stochastic Universal Sampling selection was used in order to allow the individuals with higher fitness values to have a higher probability of selection (probabilities have been assigned through Fitness Proportional selection based on sigma scaling). We take the parents 2 by 2 and we perform crossover, keeping the order the same in the parents array. We select pop_size parents because we have to keep the population size constant.

Crossover (lines 137 – 175):

To crossover the parents' genes we use uniform crossover to give equal probabilities to each gene to be passed onto the children. The crossover probability is 0.8 (80%). If the crossover is not performed, we perform asexual crossover, where each child will take the parent's gene directly. In the end we will generate pop_size children in order to keep the population size constant.

Mutation (lines 179 – 212):

The chosen mutation method is creep method which we will apply only on the genes that satisfy the mutation probability (of 0.1 (10%)). If the probability isn't met for any of the genes then the child will remain the same. We apply bounds to the creep method accordingly. For $x[j]$ the upper bound is $c[j]$ or $\max\ \_bill\_amounts[j]$ and the lower bound is 0.

Next generation selection (lines 216 – 228):

To advance the generation, we use the elitism method in which we compare the best parent (the parent with the highest fitness value) to the best child (the child with the highest fitness value). If the best parent is better than the best child, we randomly replace a child with the best parent. Otherwise, the children population remains the same.

GA Termination Conditions:

The genetic algorithm will try to compute the best individual through a certain number of $max\_iterations$. However, the GA will stop and retain the current best solution even if it is not the best global solution in certain conditions as follows:

- if the number of generations equals the number of $max\_iterations$

```
iteration == parameters.max_iterations
```

- if the max fitness value of the population remains constant over $max_{iterations}$ / 4 generations

```
nrm == int(parameters.max_iterations / 4)
```

Where $nrm = number\ of\ generations\ with\ the\ same\ max\ fitness\ value$

The GA Solution:

The solution of the genetic algorithm will be an individual from the last generation that has the best fitness value (not necessarily the best global solution) and is represented in the manner mentioned above. At the end of the GA, we will display the best individual from the last generation as well as its' fitness value.

Disclaimer:

All constant parameters of the problem have been added to a Parameters structure:

```
@dataclass
class Parameters:
    pop_size: int = 100
    n: int = 7
    max_iterations: int = 1000
    to_pay: int = 50  # sum we have to pay
    bill_values: list = field(default_factory=lambda: [])
    max_bill_amounts: list = field(default_factory=lambda: [])
    crossover_probability: float = 0.8
    mutation_probability: float = 0.1

parameters = Parameters()

parameters.bill_values = [2, 3, 5, 7, 10, 11, 13]
parameters.max_bill_amounts = [8, 11, 2, 3, 5, 7, 4]
```

However the input data can be changed when running the problem:

```
size = (int)(input("Population size ( > 0 ): "))
parameters.pop_size = size
it = (int)(input("\nMaximum number of iterations:"))
parameters.max_iterations = it
cp = (float)(input("\nCrossover probability (between 0 and 1):"))
parameters.crossover_probability = cp
mp = (float)(input("\nMutation probability (between 0 and 1):"))
parameters.crossover_probability = mp
```