Genetic Algorithm with Python

Knapsack Problem

Dr.Sherif Saad

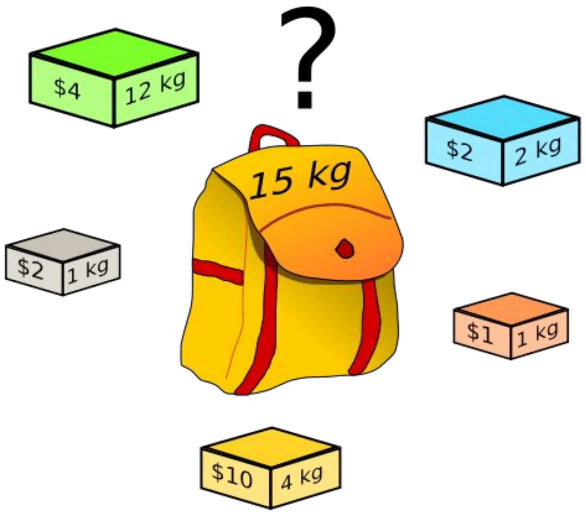
Outlines

- Problem Definition
- Input Data and Genetic Parameters
- Evolution Process
- Problem Encoding
- Initialize Population
- Parents Selection
- Crossover
- Mutation

0-1 Knapsack Problem: Definition

Given weights and values of ***n*** items, put these items in a knapsack of capacity ***W*** to get the maximum total value in the knapsack.

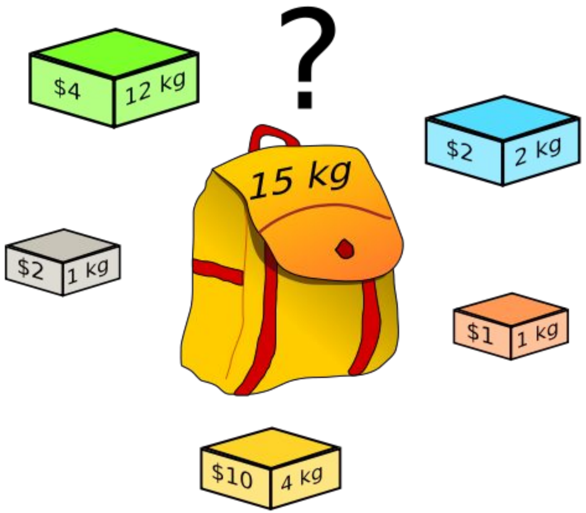
This an example of combinatorial optimization problem.



0-1 Knapsack Problem: Example

Assume we have the following set of items and a knapsack of capacity = 9. Find the items that maximize the profit with the capacity limit

Item	A	B	C	D	E	F	G
Profit	6	5	8	9	6	7	3
Weight	2	3	6	7	5	9	4



Input Data and Genetic Parameters

```
"""
we use built-in random and math modules to generate random number and map real number to integer
"""
import random
from math import floor

'''
For the knapsack problem we will use list (arrays) to store the items labels
'''

items = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

profits = [6, 5, 8, 9, 6, 7, 3]

weights = [2, 3, 6, 7, 5, 9, 4]

''' This is the weight constraint '''
knapsack_capacity = 9
```

Input Data and Genetic Parameters

```
''' The size of the initial population '''
initial_population_size = 15

''' We want 65% of  individuals in every new generation to be the result of crossover operator'''
crossover_rate = 0.65

''' The probability that a new offspring is mutated'''
mutation_rate = 0.8

''' The probability that offspring will limit the mutation operation'''
mutation_resilience = 0.9

''' We use the number of generation as termination condition '''
termination = 10


''' A list that store all the individual in the population '''
population = list()
```

Evolution Process

```
''' This the evolution process of the genetic algorithm '''
generation_counter = 0

initialize_population()

calculate_fitness()

while generation_counter < termination:

    generation_counter += 1

    ''' uss the crossover rate to decide how many new offsprings will be generated  '''
    new_offsprings = floor(len(population) * crossover_rate / 2)

    new_generation = list()

    ''' generate new offspring '''
    while new_offsprings > 0:
        ''' pick two parents '''
        parent_one = roulette_wheel()
        parent_two = roulette_wheel()

        ''' apply crossover '''
        offspring_one, offspring_two = crossover(parent_one, parent_two)

        ''' apply mutation '''
        offspring_one = mutation(offspring_one)
        offspring_two = mutation(offspring_two)

        '''store the new offsprings in a list '''
        new_generation.append(offspring_one)
        new_generation.append(offspring_two)

        new_offsprings -= 1

    ''' apply elitism and replace the weakest individuals from the current generation and add the new offsprings '''
    population = sorted(population, key=lambda k: k['profit'])
    del population[0:len(new_generation)]

    ''' this is the new generation '''
    population = new_generation + population

    '''calculate the fitness for the new generation'''
    calculate_fitness()
```

Evolution Process

```
''' This the evolution process of the genetic algorithm '''
generation_counter = 0

initialize_population()

calculate_fitness()
```

Evolution Process

```
while generation_counter < termination:

    generation_counter += 1

    ''' uss the crossover rate to decide how many new offsprings will be generated  '''
    new_offsprings = floor(len(population) * crossover_rate / 2)

    new_generation = list()

    ''' generate new offspring '''
    while new_offsprings > 0:
        ''' pick two parents '''
        parent_one = roulette_wheel()
        parent_two = roulette_wheel()

        ''' apply crossover '''
        offspring_one, offspring_two = crossover(parent_one, parent_two)

        ''' apply mutation '''
        offspring_one = mutation(offspring_one)
        offspring_two = mutation(offspring_two)

        '''store the new offsprings in a list '''
        new_generation.append(offspring_one)
        new_generation.append(offspring_two)

        new_offsprings -= 1

    ''' apply elitism and replace the weakest individuals from the current generation and add the new offsprings '''
    population = sorted(population, key=lambda k: k['profit'])
    del population[0:len(new_generation)]

    ''' this is the new generation '''
    population = new_generation + population

    '''calculate the fitness for the new generation'''
    calculate_fitness()
```

Evolution Process

```
while generation_counter < termination:

    generation_counter += 1

    ''' uss the crossover rate to decide how many new offsprings will be generated  '''
    new_offsprings = floor(len(population) * crossover_rate / 2)

    new_generation = list()
```

Evolution Process

```
''' generate new offspring '''
while new_offsprings > 0:
    ''' pick two parents '''
    parent_one = roulette_wheel()
    parent_two = roulette_wheel()

    ''' apply crossover '''
    offspring_one, offspring_two = crossover(parent_one, parent_two)

    ''' apply mutation '''
    offspring_one = mutation(offspring_one)
    offspring_two = mutation(offspring_two)

    '''store the new offsprings in a list '''
    new_generation.append(offspring_one)
    new_generation.append(offspring_two)

    new_offsprings -= 1
```

Evolution Process

```
''' apply elitism and replace the weakest individuals from the current generation and add the new offsprings '''
population = sorted(population, key=lambda k: k['profit'])
del population[0:len(new_generation)]

''' this is the new generation '''
population = new_generation + population

'''calculate the fitness for the new generation'''
calculate_fitness()
```

Problem Encoding

```
def create_chromosome():
    """
    :return: a binary string (list) chromosome that represent one candidate solution
    """
    ''' Generate a random integer between 0 and the maximum possible solution 2^n, where n is the number of items'''
    candidate = random.randint(0, 2 ** len(items) - 1)

    ''' Convert this  integer number into a binary string and append 0 bits to the left side
        if the string length less than n
    '''
    chromosome_string = format(candidate, 'b').zfill(len(items))

    ''' Convert the binary string to an array of characters'''
    chromosome = list(chromosome_string)

    return chromosome
```

Population Initialization

```
def initialize_population():
    """
     this method create the individuals for the first generation of the population
    """
    count = 0
    while count < initial_population_size:

        chromosome = create_chromosome()
        '''each individual in the population store the solution (chromosome), the weight, the fitness (profit) '''
        individual = dict()
        individual['chromosome'] = list(chromosome)

        '''for each new individual we set the profit and the weight to -1 until we calculate the fitness '''
        individual['profit'] = -1
        individual['weight'] = -1

        population.append(individual)
        count += 1
```

Calculate Fitness

```python
def calculate_fitness():
    """
     this method calculate the fitness for each individual in the population

    """
    for individual in population:
        chromosome = individual['chromosome']
        ''' if the individual survived from previous generation we do not calculate its fitness '''
        if knapsack_capacity > individual['weight'] > -1:
            continue

        ''' The individual is created in the current generation and we have to calculate its fitness '''
        fitness = 0
        weight = knapsack_capacity + 1

        ''' We calculate the fitness for the individual and make sure it obeys the weight constraint '''
        while weight > knapsack_capacity:
            weight = 0
            fitness = 0

            for g in range(0, len(chromosome)):
                fitness += int(chromosome[g]) * profits[g]
                weight += int(chromosome[g]) * weights[g]

            if weight > knapsack_capacity:
                gene = random.randint(0, len(chromosome) - 1)
                if chromosome[gene] == '1':
                    chromosome[gene] = '0'

        individual['profit'] = fitness
        individual['weight'] = weight
        individual['chromosome'] = chromosome
```

Calculate Fitness

```
"""
 this method calculate the fitness for each individual in the population

"""
for individual in population:
    chromosome = individual['chromosome']
    ''' if the individual survived from previous generation we do not calculate its fitness '''
    if knapsack_capacity > individual['weight'] > -1:
        continue
```

Calculate Fitness

```
''' The individual is created in the current generation and we have to calculate its fitness '''
fitness = 0
weight = knapsack_capacity + 1

''' We calculate the fitness for the individual and make sure it obeys the weight constraint '''
while weight > knapsack_capacity:
    weight = 0
    fitness = 0

    for g in range(0, len(chromosome)):
        fitness += int(chromosome[g]) * profits[g]
        weight += int(chromosome[g]) * weights[g]

    if weight > knapsack_capacity:
        gene = random.randint(0, len(chromosome) - 1)
        if chromosome[gene] == '1':
            chromosome[gene] = '0'

individual['profit'] = fitness
individual['weight'] = weight
individual['chromosome'] = chromosome
```

Fitness Proportionate Selection

```python
def roulette_wheel():
    """
     This method implement the fitness proportionate selection using the roulette wheel selection method
    """
    wheel = 0

    ''' the size of the wheel equal the total sum of the fitness values for the entire population '''
    for individual in population:
        wheel += individual['profit']

    ''' we set a fixed point on the wheel circumference by picking a random number between 0 and the wheel size '
    selection_point = random.randint(0, wheel)

    rotate = 0
    '''
        we start rotating the wheel by summing the fitness of each individual in the population and we stop when
        the sum is greater than or equal the fixed point
    '''
    for individual in population:
        rotate += individual['profit']
        if rotate >= selection_point:
            return individual
```

Reproduction Using Crossover

```
def crossover(parent_a, parent_b):
    """
     This method implement a one point crossover technique for binary encoding
    """

    ''' pick a random point between 0 and length of the chromosome '''
    single_point = random.randint(0, len(items) - 1)

    ''' switch the head and tail of the two parents to create two new offspring '''
    offspring_a = parent_a['chromosome'][:single_point] + parent_b['chromosome'][single_point:]

    offspring_b = parent_b['chromosome'][:single_point] + parent_a['chromosome'][single_point:]

    ''' set the weight and the profit for the new offsprings'''
    individual_one = dict()
    individual_one['chromosome'] = offspring_a
    individual_one['profit'] = -1
    individual_one['weight'] = -1

    individual_two = dict()
    individual_two['chromosome'] = offspring_b
    individual_two['profit'] = -1
    individual_two['weight'] = -1

    ''' return the new offsprings'''
    return individual_one, individual_two
```

Reproduction Using Crossover

```python
def crossover(parent_a, parent_b):
    """
     This method implement a one point crossover technique for binary encoding
    """

    ''' pick a random point between 0 and length of the chromosome '''
    single_point = random.randint(0, len(items) - 1)

    ''' switch the head and tail of the two parents to create two new offspring '''
    offspring_a = parent_a['chromosome'][:single_point] + parent_b['chromosome'][single_point:]

    offspring_b = parent_b['chromosome'][:single_point] + parent_a['chromosome'][single_point:]
```

Reproduction Using Crossover

```
''' set the weight and the profit for the new offsprings'''
individual_one = dict()
individual_one['chromosome'] = offspring_a
individual_one['profit'] = -1
individual_one['weight'] = -1

individual_two = dict()
individual_two['chromosome'] = offspring_b
individual_two['profit'] = -1
individual_two['weight'] = -1

''' return the new offsprings'''
return individual_one, individual_two
```

Reproduction Using Mutation

```python
def mutation(individual):
    """
     This method implement the bit flip mutation
    """

    ''' generate a  uniform random value as the mutation chance '''
    mutation_chance = random.uniform(0, 1)

    ''' not all the offspring will be mutated '''
    if mutation_chance < mutation_rate:

        chromosome = individual['chromosome']

        ''' if the offspring will be mutated it can only mutated twice with a resilience
            of 90% for the second mutation
        '''
        bit_flip_rate = random.uniform(0, 1)

        gene_to_mutate = random.randint(0, len(chromosome) - 1)

        if chromosome[gene_to_mutate] == '0':
            chromosome[gene_to_mutate] = '1'
        else:
            chromosome[gene_to_mutate] = '0'

        if bit_flip_rate >= mutation_resilience:
            gene_to_mutate = random.randint(0, len(chromosome) - 1)
            if chromosome[gene_to_mutate] == '0':
                chromosome[gene_to_mutate] = '1'
            else:
                chromosome[gene_to_mutate] = '0'

        individual['chromosome'] = chromosome

    return individual
```

Reproduction Using Mutation

```
"""
 This method implement the bit flip mutation
"""

''' generate a  uniform random value as the mutation chance '''
mutation_chance = random.uniform(0, 1)

''' not all the offspring will be mutated '''
if mutation_chance < mutation_rate:
```

Reproduction Using Mutation

```python
''' not all the offspring will be mutated '''
if mutation_chance < mutation_rate:

    chromosome = individual['chromosome']

    ''' if the offspring will be mutated it can only mutated twice with a resilience
        of 90% for the second mutation
    '''
    bit_flip_rate = random.uniform(0, 1)

    gene_to_mutate = random.randint(0, len(chromosome) - 1)

    if chromosome[gene_to_mutate] == '0':
        chromosome[gene_to_mutate] = '1'
    else:
        chromosome[gene_to_mutate] = '0'
```

Reproduction Using Mutation

```
    if bit_flip_rate >= mutation_resilience:
        gene_to_mutate = random.randint(0, len(chromosome) - 1)
        if chromosome[gene_to_mutate] == '0':
            chromosome[gene_to_mutate] = '1'
        else:
            chromosome[gene_to_mutate] = '0'

    individual['chromosome'] = chromosome

return individual
```

Final Output

```python
''' When the genetic algorithm terminate sort the solutions by fitness (profit) and display the result '''
population = sorted(population, key=lambda k: k['profit'])

for i in population:
    print(i)
```

Questions