Nome: Fábio Serra Vasconcelos

Sistemas para Internet 2021

TÓPICOS ESPECIAIS EM SISTEMAS PARA INTERNET

O projeto contou com 2 entidades: Individual e Item. Contendo em seu todo a maior parte do código. Passaremos pelos arquivos descrevendo suas principais partes.

Item

```
class Item:       fabioserrasv, last month • + Bag Problem
  def __init__(self, name, value, volume) -> None:
    self.__name = name
    self.__value = value
    self.__volume = volume

  def get_value(self):
    return self.__value

  def get_volume(self):
    return self.__volume
```

Figura 1 – Classe Item

Esta classe representa nossos produtos no problema da mochila, contendo informações uteis para resolução do problema como valor e volume.

Individual

```
class Individual:
  def __init__(self, items, limit, generation = 0, mutation_rate = 5, chromossomes = None) -> None:
    self.__items = items
    self.__limit = limit
    self.__mutation_rate = mutation_rate
    self.__generation = generation
    self.__value = 0
    self.__volume = 0
    self.__score = 1

    if chromossomes is None:
      self.__chromossomes = [None] * len(items)
      self.__generate_chromossomes()
    else:
      self.__chromossomes = chromossomes
```

Figura 2 – Classe Individual

Importantes variáveis são declaradas nesta classe, nela recebemos os itens (produtos) que serão usados durante nossa resolução, o limite de peso que podemos levar na mochila, o número da geração atual do Individual, a taxa de mutação que será usada no seu processo de mutação e crossover, e enfim seus cromossomos. A

declaração de seus cromossomos é feita de forma que podem ser recebidos por parâmetro ou gerados automaticamente (figura 3) caso não seja definido na instancia.

```
def __generate_chromossomes(self):
  self.__chromossomes = [zero_or_one(i) for i in self.__items]
```

```
import random
zero_or_one = lambda _x: random.choice([0, 1])
```

Figura 3 – Geração de cromossomos.

As principais e mais importantes partes do algoritmo genético são as funções de crossover e mutação, já que nelas poderemos fazer uma mescla e geração de novos indivíduos com soluções possivelmente melhores.

Mutação

Consiste inverter o valor de um cromossomo com base numa taxa de acerto, por exemplo: se tivermos uma taxa de mutação de 5% um número de 1 a 100 é sorteado, caso o número sorteado esteja no intervalo de 1% a 5% o valor do cromossomo será invertido.

```
def mutation(self, chromossomes):
  for i in range(len(chromossomes)):
    if random.randrange(0, 1000) < self.__mutation_rate:
      chromossomes[i] = 0 if chromossomes[i] == 1 else 1
  return chromossomes
```

Figura 4 – Função de mutação.

A taxa de mutação é algo que dita muito a consistência do gráfico e poderemos ver mais a respeito de sua importância na sessão de analise de resultados.

Crossover

É aqui que mesclamos os resultados, para realizarmos um crossover é necessário 2 ou mais indivíduos para realizar uma mescla em seus cromossomos para obter um conjunto de cromossomos novo.

Podemos realizar esta mescla de diversas formas dentro do nosso algoritmo, por exemplo podemos juntar os indivíduos cortando seus cromossomos pela metade e mesclando com a outra metade, também podemos sortear um número aleatório para informar onde será feita essa quebra e não ser fixada pela metade, também podemos realizar crossovers de mais que 1 ponto, por exemplo podemos sortear 4 pontos para cortar os cromossomos de forma diferente. Neste projeto implementamos 2 formas de crossovers: sorteio de número para divisão de 1 ponto, e 4 pontos para mescla.

```python
def crossover(self, other_individual):
  c1,c2 = self.__random_point(other_individual) # SORTEADO PONTO
  # c1,c2 = self.__four_points(other_individual) # 4 PONTOS

  return Individual(
    self.__items,
    self.__limit,
    self.__generation + 1,
    self.__mutation_rate,
    c1
  ), Individual(
    self.__items,
    self.__limit,
    self.__generation + 1,
    self.__mutation_rate,
    c2
  )

def __random_point(self, other_individual):
  crossover_point = random.randrange(0, len(self.__items), 1)
  chromossomes2 = other_individual.get_chromossomes()

  c1 = self.mutation(self.__chromossomes[:crossover_point] + chromossomes2[crossover_point:])
  c2 = self.mutation(chromossomes2[:crossover_point] + self.__chromossomes[crossover_point:])

  return c1, c2

def __four_points(self, other_individual):
  points = sorted(random.sample(range(len(self.__items)), 4))
  chromossomes2 = other_individual.get_chromossomes()

  c1 = self.mutation(self.__chromossomes[:points[0]] +
                     chromossomes2[points[0]:points[1]] +
                     self.__chromossomes[points[1]:points[2]] +
                     chromossomes2[points[2]:points[3]] +
                     self.__chromossomes[points[3]:])

  c2 = self.mutation(chromossomes2[:points[0]] +
                     self.__chromossomes[points[0]:points[1]] +
```

Figura 5 – Funções de crossover.

Main (Algoritmo Genético)

Tudo é utilizado no arquivo principal, que recebe todos os parâmetros para solucionar o problema.

```
class GeneticAlgorithm:
  @staticmethod
  def execute(population_length, amount_generation, bag_volume, mutation_rate):
    individuals = OrderedVector(population_length * amount_generation, revert(compare))
    best_individuals = []

    for x in range(population_length):
      individuals.insert(Individual(products, bag_volume, 0, mutation_rate))

    for i in range(amount_generation):
      # best_individuals.insert(individuals.get(0))
      best_individuals.append(individuals.get(0))
      child_individuals = OrderedVector(population_length * amount_generation, revert(compare))

      population_score = GeneticAlgorithm.get_population_sum(individuals)
      new_individuals = GeneticAlgorithm.get_new_random_population(individuals, population_score)

      for y in range(population_length // 2):
        index = y
        index2 = y+1

        i1 = new_individuals[index]
        i2 = new_individuals[index2]

        f1, f2 = i1.crossover(i2)
        child_individuals.insert(f1)
        child_individuals.insert(f2)
      individuals = child_individuals
    best = best_individuals[0]
    for i in best_individuals:
      if i.get_solution_rating() > best.get_solution_rating():
        best = i
    return best_individuals, best
```

Figura 6 – Algoritmo Genético.

Os indivíduos iniciais são criados com o número de população indicado nos parâmetros. E então é iniciado o laço de repetição das gerações também indicada nos parâmetros. A melhor solução de cada geração é inserida no Array de best_individuals. Durante essas repetições novas populações são geradas com a função de embaralhamento de indivíduos e crossover entre os indivíduos dessas populações geradas, até terminar o laço de repetição.

```python
@staticmethod
def get_population_sum(population):
  return sum([x.get_solution_rating() for x in population])

@staticmethod
def russian_roullete(population, population_score):
  random_value = random.randrange(0,math.floor(population_score))
  sum = 0
  for i in population:
    if sum + i.get_solution_rating() >= random_value:
      return i
    sum += i.get_solution_rating()

def get_new_random_population(population, population_score):
  return [
    GeneticAlgorithm.russian_roullete(population, population_score)
    for _ in range(len(population))
  ]      fabioserrasv, 3 weeks ago • + Gerando nova população pela r
```

Figura 7 – Funções de embaralhamento e avaliação de população.

Resultados

Foi realizado 2 rodadas de testes priorizando a taxa de mutação e número de gerações. Ao analisar os gráficos é possível concluir que quanto menor a taxa de mutação o resultado é visivelmente melhor. E o maior número de gerações também melhora a solução mas números muito grandes aumentam o tempo de execução consideravelmente.

```python
DATA_TESTS = [
  # BAG_VOLUME, MUTATION_RATE, POPULATION_LENGTH, GENERATION
  [20, 2, 100, 200],
  [20, 5, 100, 200],
  [20, 10, 100, 200],
  [20, 50, 100, 200],
]
```

Dados de teste 1

Bag Volume: 20

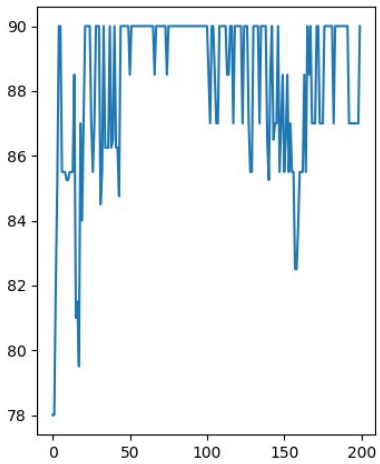
Mutation Rate: 2

Population Length: 100

Gerações: 200

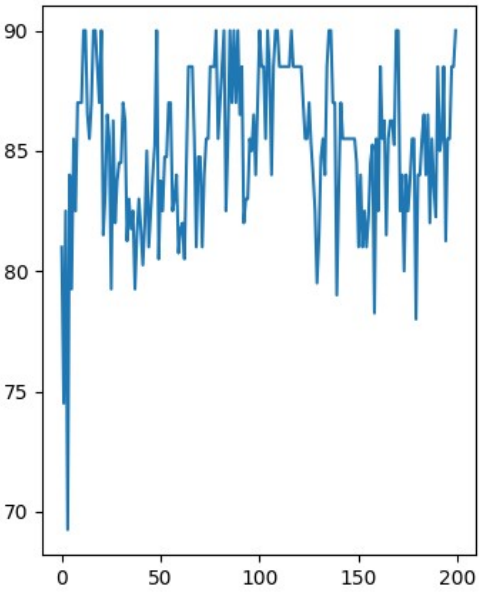
Solução Valor: 93.04

Solução Peso:19.081



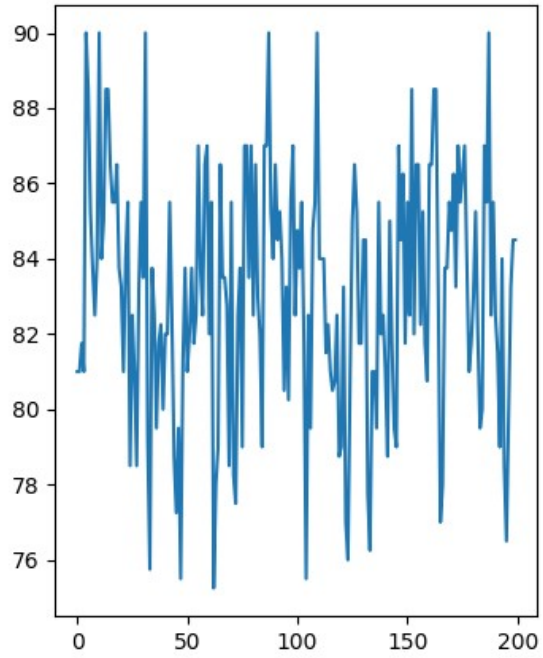
Dados de teste 2

Bag Volume: 20
Mutation Rate: 5
Population Length: 100
Gerações: 200
Solução Valor: 92.58000000000001
Solução Peso:19.67277



Dados de teste 3

Bag Volume: 20
Mutation Rate: 10
Population Length: 100
Gerações: 200
Solução Valor: 92.58000000000001
Solução Peso:19.67277



Dados de teste 4

Bag Volume: 20

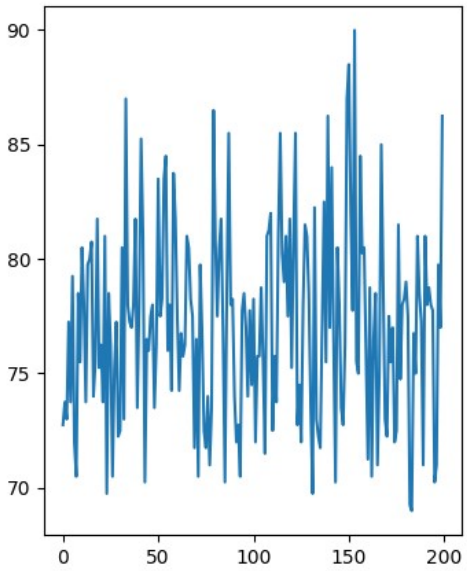
Mutation Rate: 50

Population Length: 100

Gerações: 200

Solução Valor: 87.13000000000001

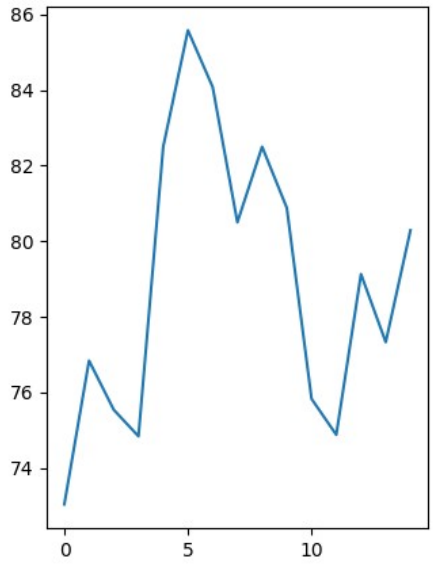
Solução Peso:19.95409



```
DATA_TESTS = [
  # BAG_VOLUME, MUTATION_RATE, POPULATION_LENGTH, GENERATION
  [20, 2, 100, 15],
  [20, 2, 100, 200],
  [20, 5, 100, 15],
  [20, 5, 10, 200],
]
```

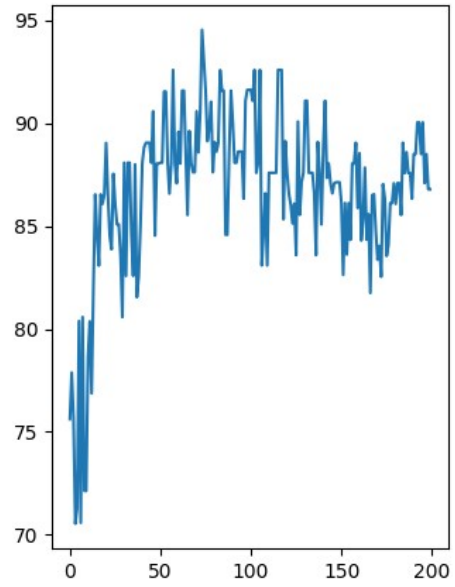
Dados de teste 1

Bag Volume: 20
Mutation Rate: 2
Population Length: 100
Gerações: 15
Solução Valor: 88.8
Solução Peso:18.84132



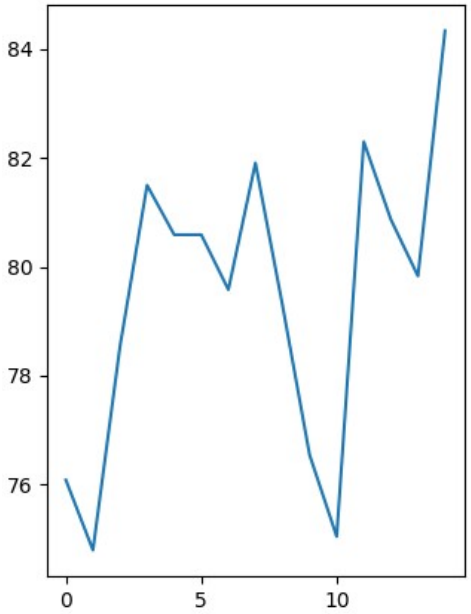
Dados de teste 2

Bag Volume: 20
Mutation Rate: 2
Population Length: 100
Gerações: 200
Solução Valor: 93.12000000000002
Solução Peso:19.78454



Dados de teste 3

Bag Volume: 20
Mutation Rate: 5
Population Length: 100
Gerações: 15
Solução Valor: 88.63000000000001
Solução Peso:19.41409



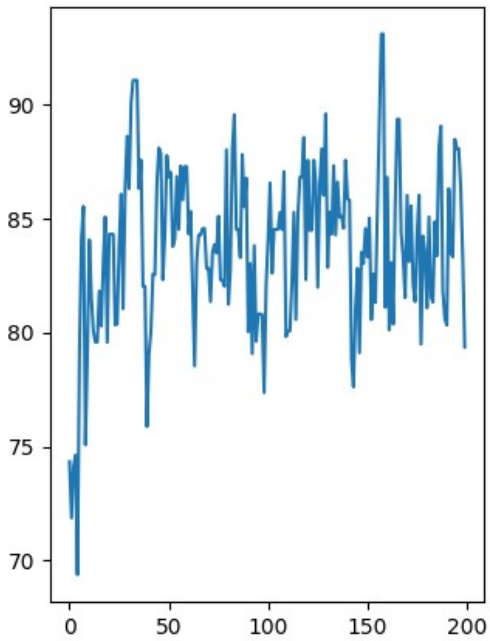
Dados de teste 4

Bag Volume: 20
Mutation Rate: 5
Population Length: 10

Gerações: 200

Solução Valor: 91.58000000000001

Solução Peso:19.54277



Conclusão

Para este problema da mochila acredito que a melhor configuração seja um número de taxa de mutação baixo até mesmo sendo considerado na casa de 0,0X%. Mantendo um número alto de gerações e o número de população.

Mas acredito que essa analise não explique completamente o algoritmo genético, já que estamos limitando seu funcionamento no problema da mochila, nessas situações podemos ver que a taxa de mutação baixa e um número considerável de população e gerações levam a bons resultados.

Passando pelos parâmetros: A taxa de mutação parece trabalhar melhor com um número baixo entre 1-5, trazendo melhores resultados. Já o tamanho da população é o inverso, trabalhando melhor com números maiores, tendo mais indivíduos no conjunto. O número de gerações é basicamente quantas repetições teremos para procurar soluções, ou seja, quanto maior mais a possibilidade de achar a melhor solução mas também maior o tempo de execução.