
FIAP

PYTHON – AULA 8
LÓGICA DE PROGRAMAÇÃO APLICADA

AGENDA

Aula 1	Instalando e conhecendo o Python e as ferramentas do curso. Revisitando lógica de programação em Python.
Aula 2	Revisitando lógica de programação em Python – continuação.
Aula 3	Trabalhando com listas, tuplas e dicionários. Leitura e escrita de JSON.
Aula 4	Leitura e escrita de arquivos. Introdução a Numpy e Pandas.

AGENDA

Aula 5	Orientação a objeto no Python.
Aula 6	Trabalhando com os algoritmos. Ordenação e Recursão. – Pesquisa em largura.
Aula 7	Algoritmo de Dijkstra.
Aula 8	Algoritmos Genéticos.

PROGRAMANDO EM PYTHON

OTIMIZAÇÃO

ALGORITMOS GENÉTICOS

Introdução

- Voltemos ao problema do caixeiro-viajante.

- Vimos que para visitar 100 cidades teríamos 10^{158} opções; considerando 1 bilhão de soluções por segundo, levaríamos 10^{140} anos para encontrar a melhor rota (mais tempo do que a idade do universo).

- Esse tipo de problema é denominado **intratável**, ou seja, quando conseguirmos esgotar todas as opções, a solução já não fará mais sentido.

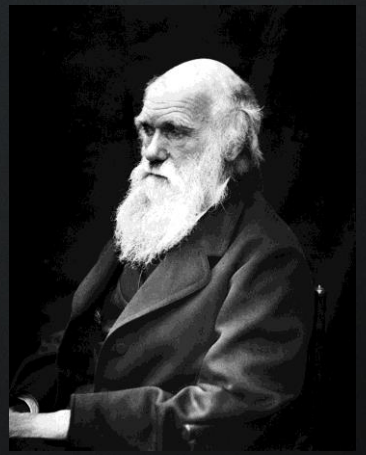
Introdução

- Mas e se, ao invés de esgotar todas as opções para encontrar a melhor rota, pudéssemos buscar uma aproximação?

Os algoritmos genéticos são técnicas **heurísticas** de aproximação, que devem ser utilizados quando abordagens tradicionais não são apropriadas para se chegar a resolução de um problema em um tempo satisfatório.

A **heurística** é um processo de pensamento pela busca de soluções para questões complexas de modo incerto e incompleto.

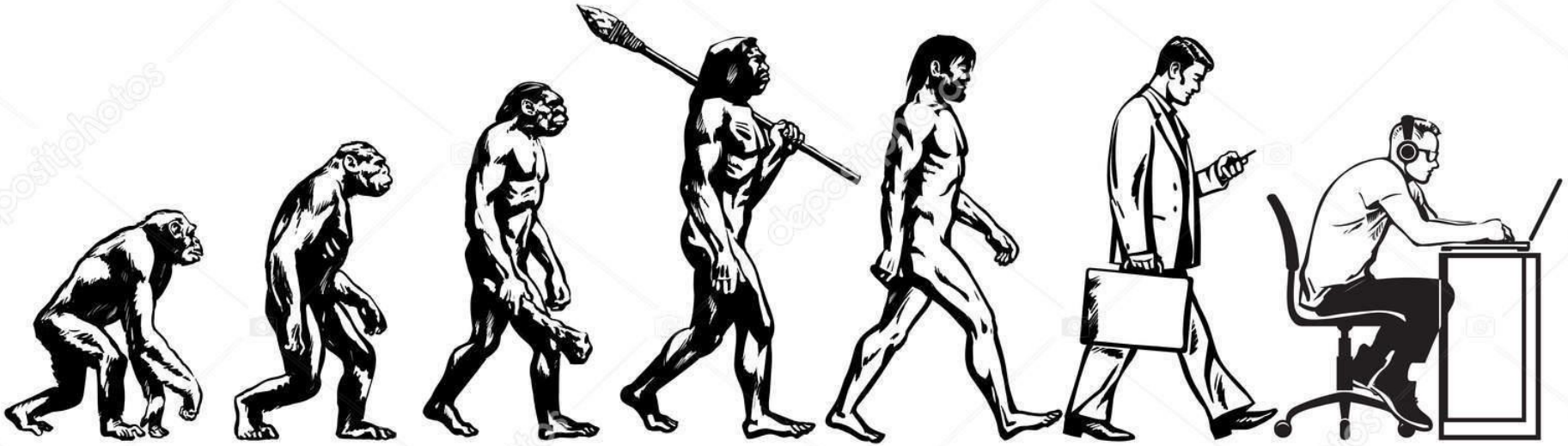
Background em biologia

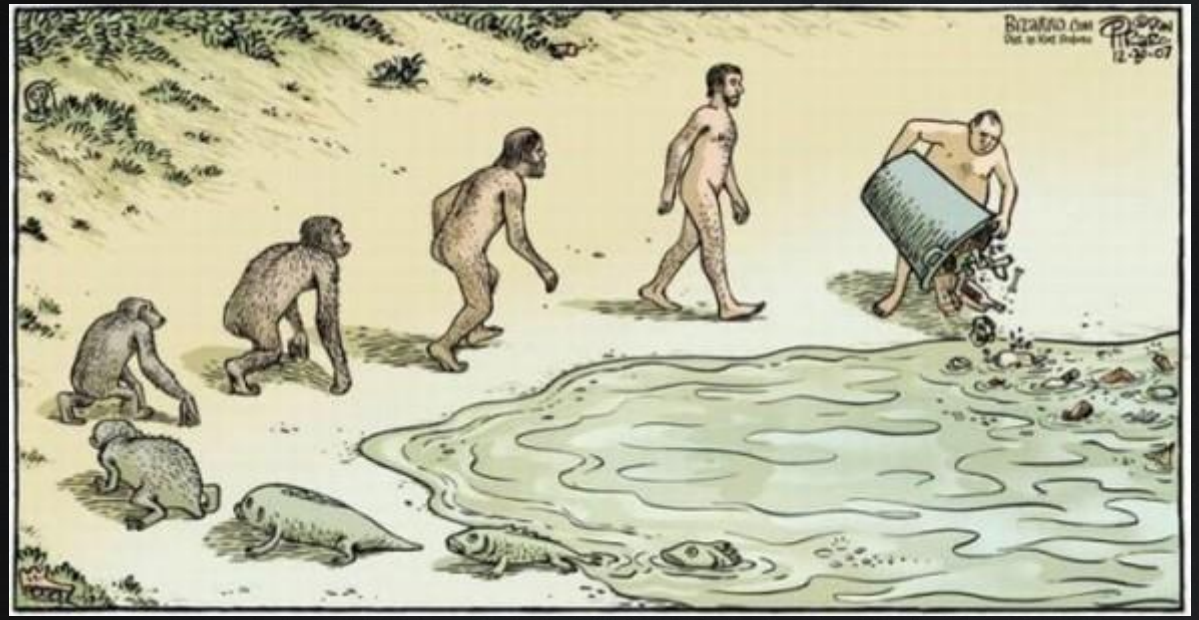


- Em 1859 o naturalista britânico Charles Darwin, apresenta a Teoria da Evolução, onde esboça a origem da espécies por meio da seleção natural, adaptação, preservação e propagação dos mais favorecidos pela luta na vida.

- A ideia central da teoria científica de Darwin é a de que os seres vivos possuem uma ancestralidade comum, que com o passar de muitos milhões de anos foram dando surgimento a novas espécies em função de influências causadas pelo meio ambiente, por mutações genéticas e novos cruzamentos.

- Os seres mais aptos continuam a existir e propagam seus genes para novas populações, os menos favorecidos tendem a entrar em extinção.





Extraído de: https://brainly.com.br/tarefa/13646589

Algoritmos Genéticos

- Os algoritmos genéticos, em analogia aos processos de seleção natural que ocorrem na natureza, fazem referências cruzadas e vão descartando as soluções ruins, aproveitando as boas em uma rede que vai sendo otimizada conforme o uso.

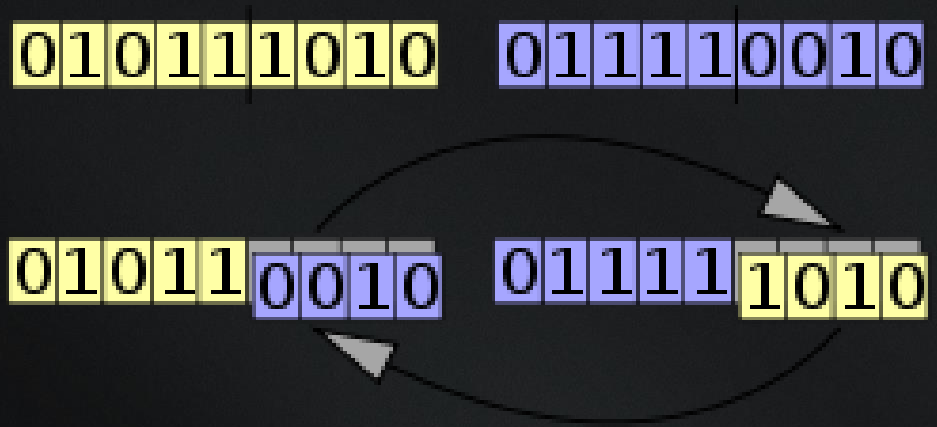
- O algoritmo inicia com uma população representada por "cromossomos".

- A população é uma configuração inicial para determinado problema e pode iniciar de forma randômica.

Algoritmos Genéticos

- Com o tempo a população evolui passando por **mutações** e propagando sua informações para outras.

- Os **cromossomos** podem ser representações **binárias**, conforme exemplo da imagem abaixo.



Extraído de: https://pt.wikipedia.org/wiki/Ficheiro:Computational.science.Genetic.algorithm.Crossover.One.Point.svg

Algoritmos Genéticos

- Nesse exemplo de corte simples, os cromossomos pais originaram dois filhos por meio da combinação dos quatro últimos bits de cada pai.

- Os Algoritmos genéticos são sistemas probabilísticos e não determinísticos, desse modo, apresentam um conjunto de soluções possíveis e não apenas uma solução isolada para determinado problema.

- As soluções mais apropriadas são selecionadas para formar as novas populações que serão cruzadas favorecendo a evolução da solução.

- São os mais utilizados na logística, pois são muito empregados em problemas de busca e seleção, porém de forma distribuída e não sequencial.

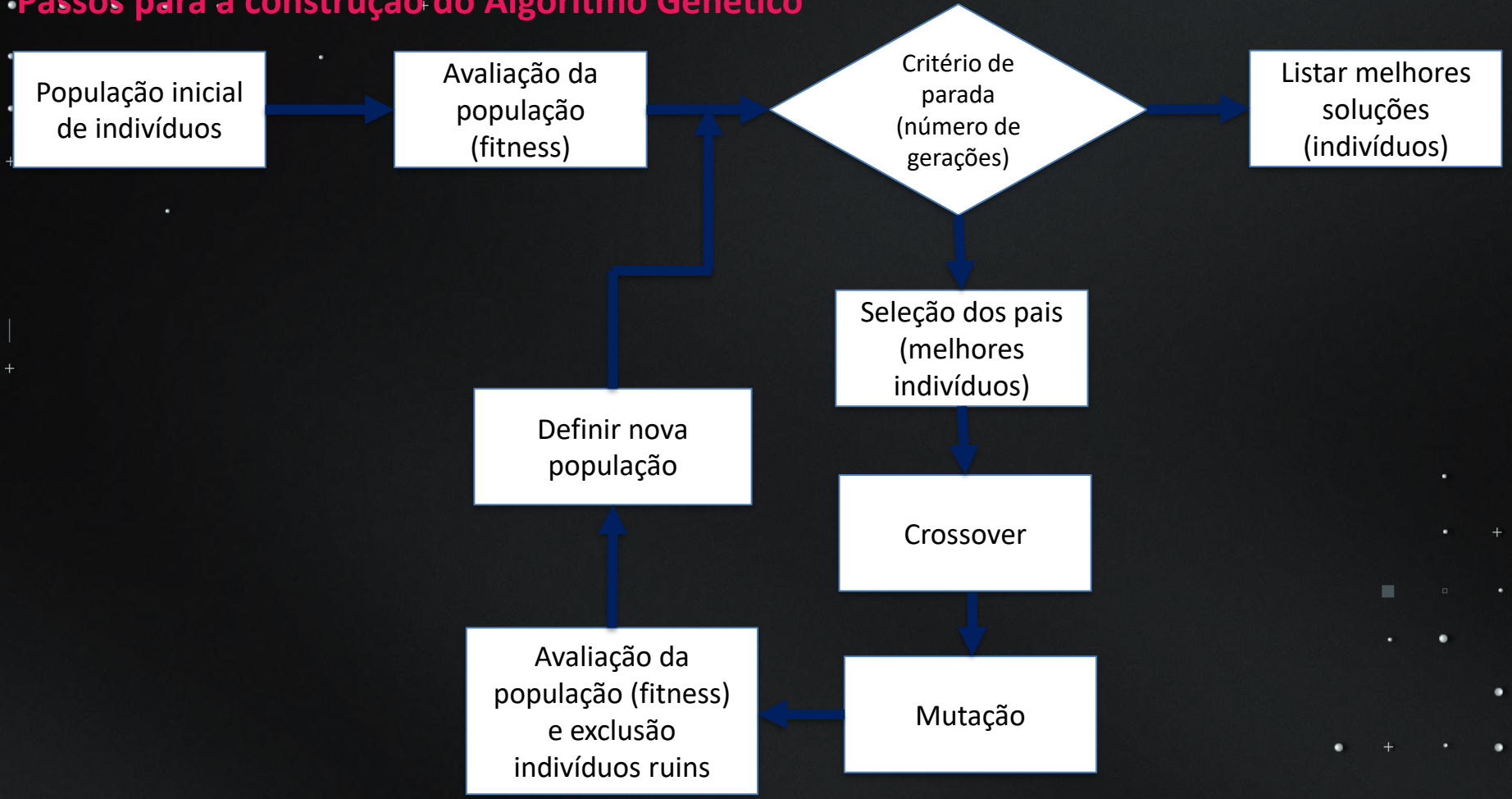
Aplicações

- Agendamento de passagens
- Otimização de espaço
- Roteirização
- Distribuição eficiente de mercadorias
- Fluxo de caixa inteligente
- Classificação de Clientes (Data Mining)
- Análise de alternativas de investimento em Projetos
- Otimização de layout de circuitos
- Otimização em Redes Neurais Artificiais e Lógica Fuzzy

Passos para a construção do Algoritmo Genético

1. Criar a população inicial de indivíduos (cromossomos) de forma randômica
2. Avaliar a população; custo de cada solução (indivíduo) – função fitness
3. Definir critério de parada - número de gerações
4. Selecionar indivíduos para reprodução – melhores pais
5. Aplicar operadores de recombinação (crossover) e mutação.
6. Avaliar os novos indivíduos e adicioná-los a população.
7. Excluir velhos membros da população – indivíduos ruins
8. Retornar ao passo 2 até encontrar a melhor solução.

Passos para a construção do Algoritmo Genético



População inicial

- A população é composta pelas soluções propostas ou uma lista de vetores.
- Os vetores são os indivíduos da população, sendo estes compostos por genes.
- Os genes são os dados (objetos) de cada indivíduo ou cromossomo.
- Exemplo do problema da mochila

Capacidade máxima
de carga W = 10kg

Objeto	Valor	Peso (kg)
Lanterna	80	1
Faca	85	1
Roupas	90	3
Livros	50	0.5
Saco de dormir	105	2
Remédios	90	2
Alimentos	200	4
Rede de descanso	100	2.5

População inicial – Matriz binária

Cromossomo

	Lanterna	Faca	Roupas	Livros	Saco	Remédios	Alimentos	Rede	Valor total
X_{11}	1	0	1	0	0	1	1	0	460
X_{21}	1	1	0	0	1	1	1	0	560
X_{31}	0	1	1	1	0	1	0	1	415
X_{41}	1	0	0	1	0	0	1	1	430
X_{51}	1	1	1	1	0	1	0	1	495

Gene

Função Fitness

- A função fitness vai depender do problema a ser tratado; deve ser escolhida com cuidado, pois precisa abarcar todo conhecimento sobre o problema a ser resolvido.

- Como a função de avaliação é a nota dada ao individuo na resolução do problema, neste exemplo, é a somatória do valor dos itens presentes na matriz binária.

- A avaliação da população é feita ordenando-se as maiores notas conferidas aos indivíduos.

	Lanterna	Faca	Roupas	Livros	Saco	Remédios	Alimentos	Rede	Valor total
X_{21}	1	1	0	0	1	1	1	0	560
X_{51}	1	1	1	1	0	1	0	1	495
X_{11}	1	0	1	0	0	1	1	0	460
X_{41}	1	0	0	1	0	0	1	1	430
X_{31}	0	1	1	1	0	1	0	1	415

Escolha dos pais

- A seleção dos pais deve simular a seleção natural que age sobre organismos biológicos, em que pais mais aptos geram mais filhos, entretanto, sem coibir as geração de descendentes dos pais menos aptos.

- Deve-se portanto, priorizar indivíduos com as notas mais altas, sem desprezar os demais. Isso é importante, pois gera diversidade para a população evoluir de forma satisfatória.

- Pode acontecer que alguns indivíduos que não são tão aptos a se reproduzirem, possuam genes que sejam favoráveis, e que podem estar pouco presentes em indivíduos mais aptos.

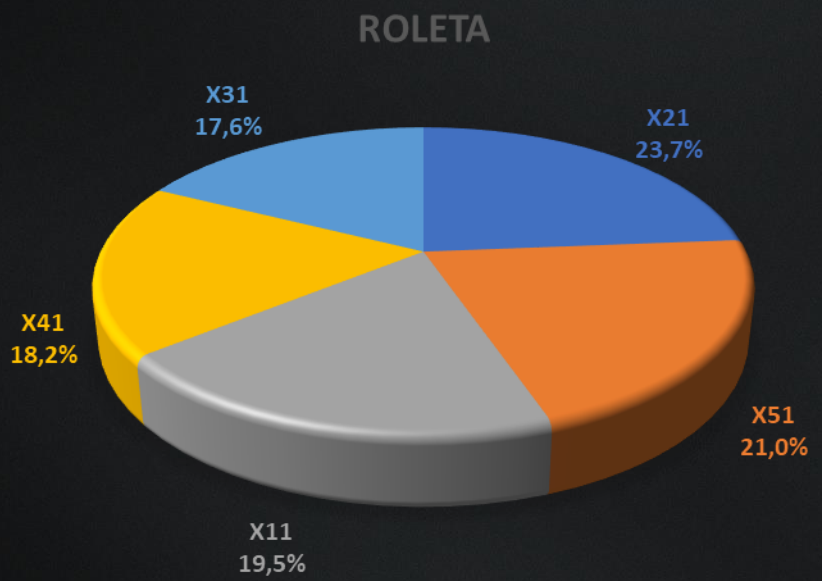
Percentual de seleção / Roleta

Para dar chance a todos os indivíduos se reproduzirem, vamos utilizar a técnica mais comumente empregada: **a roleta viciada.**

- Criamos uma roleta (virtual) na qual cada indivíduo recebe um valor proporcional a sua avaliação.
- A roleta deve ser girada de forma aleatória.
- Desse modo, os mais forte tem preferência para reprodução, porém os mais fracos também possuem chances.

Percentual de seleção / Roleta

Para o exemplo da mochila, a soma total das notas dos indivíduos é de 2360 pontos. Assim, a distribuição das probabilidades deve seguir conforme mostrado na imagem.



Reprodução (Crossover)

- Utilizamos o operador de crossover para fazer a reprodução após a escolha dos pais. Existem diferentes formas de se aplicar o crossover, sendo o mais simples o operador de um ponto.

- Este operador consiste em definir um ponto de corte entre os genes dos indivíduos para seja feita a reprodução cruzada; o ponto de corte também deve ser sorteado pela roleta.

- Considerando que cada cromossomo tem 8 genes, temos 7 possíveis pontos de corte.

1	1	0	0	1	1	1	0

Reprodução (Crossover)

- Vamos supor que tenham sidos selecionados os indivíduos X_{21} e X_{11} e que o ponto de corte escolhido foi o 2, após a execução da roleta.

X_{21}	1	1	0	0	1	1	1	0
X_{11}	1	0	1	0	0	1	1	0

Pais

- Aplicando o crossover teremos os seguintes filhos:

1	1	1	0	0	1	1	0
1	0	0	0	1	1	1	0

Filhos

Mutação

- O próximo passo consiste na aplicação da mutação.
- A mutação propicia diversidade na população para que esta escape de soluções ótimas locais e tenha maior chance de se aproximar de soluções ótimas globais.
- O operador de mutação tem associada uma probabilidade muito baixa. A taxa aplicada normalmente está na ordem de 1% a 5%.

1	1	1	0	0	1	1	0
1	1	1	0	0	1	0	0

Mutação no penúltimo gene

Aplicação no Python - Itens

- Vamos começar definindo uma classe para os itens da mochila, com os atributos nome, valor e peso.

```
class Item():
    def __init__(self, nome, valor, peso):
        self.nome = nome
        self.valor = valor
        self.peso = peso
```

Aplicação no Python - Indivíduo

Posteriormente, vamos definir uma classe para os Indivíduos.

```
class Individuo():
    def __init__(self, pesos, valores):
        self.pesos = pesos
        self.valores = valores
        self.limite_peso = 10
        self.nota_avaliacao = 0
        self.geracao = 0
        self.cromossomo = []

        for i in range(len(pesos)):
            if random() < 0.5
                    self.cromossomo.append("0")
            else:
                    self.cromossomo.append("1")
```

Temos um laço para inicializar o cromossomo com valores aleatórios para os genes.

Número randômico entre 0 e 1 com 50% de chance para cada (0.5)

Aplicação no Python - Indivíduo

Podemos agora inicializar os itens da mochila e adiciona-los a uma lista

```
if __name__ == "__main__":
    lista_itens = []
    lista_itens.append(Item("Lanterna", 80, 1))
    lista_itens.append(Item("Faca", 85, 1))
    lista_itens.append(Item("Roupas", 90, 3))
    lista_itens.append(Item("Livros", 50, 0.5))
    lista_itens.append(Item("Saco de dormir", 105, 2))
    lista_itens.append(Item("Remédios", 90, 2))
    lista_itens.append(Item("Alimentos", 200, 4))
    lista_itens.append(Item("Rede de descanso", 100, 2.5))
```

Aplicação no Python - Indivíduo

Agora podemos realizar alguns testes preliminares.

```
valores = []
pesos = []

for item in lista_itens:
    valores.append(item.valor)
    pesos.append(item.peso)

individuo_1 = Individuo(pesos, valores)
print("Pesos = %s" % str(individuo_1.pesos))
print("Valores = %s" % str(individuo_1.valores))
print("Cromossomo = %s" % str(individuo_1.cromossomo))
```

Aplicação no Python - Fitness

Primeiro acrescentamos mais um atributo na classe Individuo.

```
self.avaliacao = 0
```

Depois definimos a função de avaliação (fitness) dentro desta mesma classe.

```
def fitness(self):
    nota = 0
    soma_pesos = 0
    for i in range(len(self.cromossomo)):
        if self.cromossomo[i] == '1':
            nota += self.valores[i]
            soma_pesos += self.pesos[i]
    if soma_pesos > self.limite_peso:
        nota = 1
    self.avaliacao = nota
    self.peso_total = soma_pesos
```

Importante não descartamos a solução caso ela extrapole a carga máxima da mochila. Neste caso atribuímos um valor bem baixo; nota =1

Aplicação no Python - Fitness

Agora podemos testar a função

```
individuo_1 = Individuo(pesos, valores)
print("Indivíduo 1")
print("Pesos = %s" % str(individuo_1.pesos))
print("Valores = %s" % str(individuo_1.valores))
print("Cromossomo = %s" %
str(individuo_1.cromossomo))

individuo_1.fitness()
print("Avaliação = %s " % individuo_1.avaliacao)
print("Peso total:  = %s " %
individuo_1.peso_total)
```

Aplicação no Python - Crossover

O indivíduo que evoca a função crossover indica com qual individuo irá reproduzir

```
def crossover(self, individuo):
    ponto_corte = round(random() * len(self.cromossomo))
    filho_1 =  self.cromossomo[0:ponto_corte] + individuo.cromossomo[ponto_corte::]
    filho_2 =  individuo.cromossomo[0:ponto_corte] + self.cromossomo[ponto_corte::]
    filhos = [Individuo(self.pesos, self.valores, self.limite_peso, self.geracao + 1),
              Individuo(self.pesos, self.valores, self.limite_peso, self.geracao + 1)]

    filhos[0].cromossomo = filho_1
    filhos[1].cromossomo = filho_2
    return filhos
```

O ponto de corte recebe um número aleatório entre 0 e 1 vezes o tamanho do cromossomo; [ponto_corte::] **vai do ponto de corte até o restante do vetor.**

Aplicação no Python - Crossover

Para testar, criamos dois indivíduos, aplicamos a função fitness e a crossover, reproduzindo o indivíduo 1 com o indivíduo 2.

```
Individuo_1 = Individuo(pesos, valores)
print("Indivíduo 1")
print("Pesos = %s" % str(individuo_1.pesos))
print("Valores = %s" %
str(individuo_1.valores))
print("Cromossomo = %s" %
str(individuo_1.cromossomo))

individuo_1.fitness()
print("Avaliação = %s " %
individuo_1.avaliacao)
print("Peso total:  = %s " %
individuo_1.peso_total)

print("----------------")
```

```
Individuo_2 = Individuo(pesos, valores)
print("Indivíduo 2")
print("Pesos = %s" % str(individuo_2.pesos))
print("Valores = %s" % str(individuo_2.valores))
print("Cromossomo = %s" %
str(individuo_2.cromossomo))

individuo_2.fitness()
print("Avaliação = %s " % individuo_2.avaliacao)
print("Peso total  = %s " %
individuo_2.peso_total)

filhos = individuo_1.crossover(individuo_2)
print("Filho 1 = %s" % filhos[0].cromossomo)
print("Filho 2 = %s" %filhos[1].cromossomo)
```

Aplicação no Python - Mutação

Para o processo de mutação definimos uma taxa de 5%. Caso o valor aleatório entre 0 e 1 seja menor que a taxa de mutação fazemos a mudança do gene.

```
def mutacao(self, taxa_mutacao = 0.05):
    print("Antes da mutação = %s " % self.cromossomo)
    for i in range(len(self.cromossomo)):
        if random() < taxa_mutacao:
            if self.cromossomo[i] == '1':
                self.cromossomo[i] = '0'
            else:
                self.cromossomo[i] = '1'
    print("Depois da mutação = %s " % self.cromossomo)
    return self
```

Aplicação no Python – Geração da população inicial

Após definir os itens para a mochila e os indivíduos que apresentam as soluções, vamos implementar uma nova classe denominada Algoritmo Genético para iniciar efetivamente nosso algoritmo, começando por iniciar a população de indivíduos.

```
class AlgoritmoGenetico():
    def __init__(self):
        self.tamanho_populacao = 5
        self.geracao = 0
        self.melhor_solucao = 0
        self.populacao = []
        self.solucoes = []
```

Aplicação no Python – Geração da população inicial

Dentro da classe Algoritmo Genético vamos declarar alguns métodos, a começar pelo que inicializa a população de indivíduos.

```
def comeca_populacao(self, pesos, valores):
    for i in range(self.tamanho_populacao):
        self.populacao.append(Individuo(pesos, valores))
        self.melhor_solucao = self.populacao[0]
```

Aplicação no Python – Geração da população inicial

Para testar, inicializamos a população e fazemos um laço mostrando os indivíduos criados.

```
algoritmo_genetico = AlgoritmoGenetico()
algoritmo_genetico.inicializa_populacao(pesos,valores)
for i in range(algoritmo_genetico.tamanho_populacao):
    print("Individuo: %s" % i, "Pesos = %s" %
str(algoritmo_genetico.populacao[i].pesos),
          "Valores = %s" % str(algoritmo_genetico.populacao[i].valores),
          "Cromossomo = %s" % str(algoritmo_genetico.populacao[i].cromossomo))
```

Note que os pesos e valores são os mesmo para todos os indivíduos, porém os cromossomos são diferentes.

Aplicação no Python – Geração da população inicial

O próximo passo é ordenar a população e selecionar os melhores indivíduos. Assim, criamos mais dois métodos para a classe.

```
def ordena_populacao(self):
    self.populacao = sorted(self.populacao,
                            key = lambda populacao:
populacao.avaliacao,
                            reverse=True)

def melhor_individuo(self, individuo):
    if individuo.avaliacao > self.melhor_solucao:
        self.melhor_solucao = individuo
```

Aplicação no Python – Geração da população inicial

Vamos testar. Primeiramente, inicializamos a população e aplicamos a função fitness para cada indivíduo. Depois, ordenamos e buscamos pelo melhor indivíduo.

```
algoritmo_genetico = AlgoritmoGenetico()
algoritmo_genetico.comeca_populacao(pesos,valores)

for individuo in algoritmo_genetico.populacao:
    individuo.fitness()

algoritmo_genetico.ordena_populacao()
algoritmo_genetico.melhor_individuo(algoritmo_genetico.populacao[0])
print("Melhor solução: %s" %
algoritmo_genetico.melhor_solucao.cromossomo,
      "Nota = %s\n" % str(algoritmo_genetico.melhor_solucao.avaliacao))
```

Aplicação no Python – Geração da população inicial

Agora vamos definir um método para somar as avaliações de cada solução da população de indivíduos.

```
def total_avaliacoes(self):
    soma = 0
    for individuo in self.populacao:
        soma += individuo.avaliacao
    return soma
```

Vamos testar !

```
total_avaliacoes = algoritmo_genetico.total_avaliacoes()
print("Total: %s" % total_avaliacoes)
```

Aplicação no Python – Seleção dos pais

Para a seleção dos pais vamos simular o método da roleta viciada.

```
def roleta_viciada(self, total_avaliacao):
    pai = -1
    valor_escolhido = random() * total_avaliacao
    soma = 0
    i = 0
    while i < len(self.populacao) and soma < valor_escolhido:
        soma += self.populacao[i].avaliacao
        pai += 1
        i += 1
    return pai
```

Aplicação no Python – Novas populações

Finalmente, vamos gerar duas populações para encontrar a melhor solução. Vamos definir uma variável: `nova_populacao = []`

Criamos os filhos e adicionamos a nova população.

```
Filhos = algoritmo_genetico.populacao[pai_1].crossover(algoritmo_genetico.populacao[pai_2])
nova_populacao.append(filhos[0].mutacao())
nova_populacao.append(filhos[1].mutacao())
```

Aplicação no Python – Novas populações

Iteremos por duas gerações para encontrar a melhor solução entre as elas.

```
algoritmo_genetico.populacao = list(nova_populacao)
for individuo in algoritmo_genetico.populacao:
    individuo.fitness()
algoritmo_genetico.ordena_populacao()
algoritmo_genetico.melhor_individuo(algoritmo_genetico.populacao[0])
soma = algoritmo_genetico.total_avaliacoes()

print("Solução escolhida: %s" % algoritmo_genetico.melhor_solucao.cromossomo,
"\nValor: %s" % algoritmo_genetico.melhor_solucao.avaliacao)
```

Exercícios

1. Para finalizar a aplicação, elabore um método específico na classe algoritmo genético para gerar diversas populações e mostrar a melhor solução entre elas.

2. Separe o código em arquivos diferentes dentro do mesmo projeto

3. Ajuste os valores pré-fixados em numero de gerações, taxa de mutação, peso total para valores variáveis.

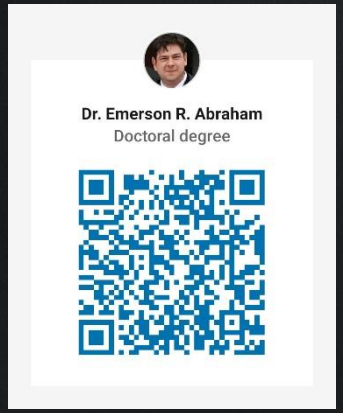
BIBLIOGRAFIA BÁSICA

- BEAZLEY, David. Python Essential Reference, 2009.
- BHARGAVA, ADITYA Y. Entendendo Algoritmos. Um guia ilustrado para programadores e outros curiosos. São Paulo: Ed. Novatec, 2017
- CORMEN, THOMAS H. et al. Algoritmos: teoria e prática. Rio de Janeiro: Elsevier, 2002
- COSTA, Sérgio Souza. Recursividade. Professor Adjunto da Universidade Federal do Maranhão.
- DOWNEY, ALLEN B. Pense em Python. Pense como um cientista da computação. São Paulo: Ed. Novatec, 2016
- GRANATYR, Jones; PACHOLOK, Edson. IA Expert Academy. Disponível em: https://iaexpert.academy/
- KOPEC, DAVID. Problemas clássicos de ciência da computação com Python. São Paulo: Ed. Novatec, 2019
- LINDEN, Ricardo. Algoritmos Genéticos. 3 edição. Rio de Janeiro: Editora Moderna, 2012
- MCKINNEY, WILLIAM WESLEY. Python para análise de dados. Tratamento de dados com Pandas, Numpy e Ipython. São Paulo: Ed. Novatec, 2018

BIBLIOGRAFIA BÁSICA

- TENEMBAUM, Aaron M. Estrutura de Dados Usando C. Sao Paulo: Makron Books do Brasil, 1995.
- VELLOSO, Paulo. Estruturas de Dados. Rio de Janeiro: Ed. Campus, 1991.
- VILLAS, Marcos V & Outros. Estruturas de Dados: Conceitos e Tecnicas de implementacao. RJ: Ed. Campus, 1993.
- PREISS, Bruno P. Estrutura de dados e algoritmos: Padrões de projetos orientados a objetos com Java. Rio de Janeiro: Editora Campus, 2001.
- PUGA, Sandra; RISSETTI, Gerson. Lógica de programação e estruturas de dados. 2016.
- SILVA, Osmar Quirino. Estrutura de Dados e Algoritmos Usando C. Fundamentos e Aplicações. Rio de Janeiro: Editora Ciência Moderna, 2007.
- ZIVIANI, N. Projeto de algoritmos com implementações em pascal e C. São Paulo: Editora Thomsom, 2002.

OBRIGADO



FIAP


FIAP