SÉRGIO POLIMANTE SOUTO

PÓS TECH

IA PARA DEVS

INTRODUÇÃO AO ALGORITMO GENÉTICO

# AULA 04

**SUMÁRIO**

## O QUE VEM POR AÍ?

Nesta aula, vamos colocar em prática o que aprendemos e escrever um código em Python, usando Pygame. Vamos escrever um código que resolve “o problema do caixeiro” viajante para n cidades. Você terá a oportunidade de entender como um indivíduo representa uma solução e como a função de aptidão e os processos de seleção, incluindo seleção aleatória dos 10 primeiros indivíduos e seleção probabilística, desempenham papéis cruciais nesse contexto. Prepare-se para uma aula interativa e aplicada, se aprofundando nas nuances da representação de indivíduos e nos processos de avaliação e seleção em Algoritmos Genéticos.

## HANDS ON

Nesta videoaula, nós vamos colocar nossos conhecimentos em prática resolvendo o “problema do caixeiro viajante”.

Vamos utilizar o Pygame para fazer a animação da solução sendo encontrada e utilizaremos a IDE Spyder distribuída pela Anaconda.

Nesta aula, também vamos focar na parte inicial do algoritmo genético: geração da população inicial, cálculo de aptidão (fitness), condição de término do algoritmo genético e método de seleção.

# SAIBA MAIS

## DEFINIÇÃO DO PROBLEMA

Antes de pensar como podemos fazer a codificação, vamos pensar na definição do nosso problema. Estamos nos propondo a resolver o problema do caixeiro viajante que é proposto da seguinte maneira:

> *O problema do caixeiro viajante é um desafio de otimização que consiste em encontrar a rota mais curta que visite um conjunto de cidades exatamente uma vez e retorne ao ponto de partida. O objetivo é minimizar a distância total percorrida, dadas as distâncias entre cada par de cidades.*

Então, começamos com um conjunto de cidades. Vamos representar cada cidade como um ponto vermelho em uma tela branca:

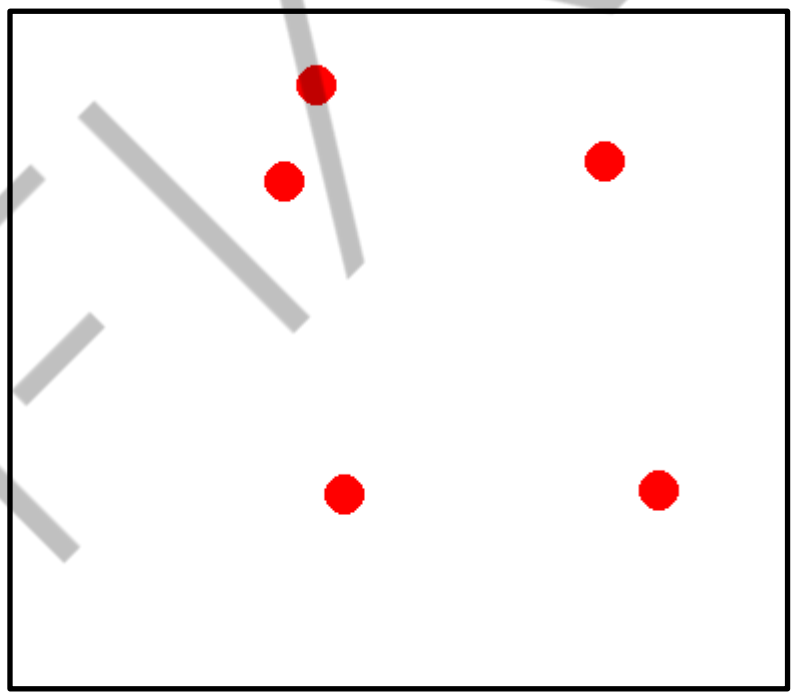

Figura 1 - Representação do problema do caixeiro viajante
Fonte: elaborado pelo autor (2024), adaptado por FIAP (2024)

A localização da cidade é dada pela sua posição de latitude (eixo y) e longitude (eixo x). Logo, podemos representar cada cidade como uma tupla com valores x e y (x, y):

- cidade 1: (733, 251).
- cidade 2: (706, 87).
- cidade 3: (546, 97).
- cidade 4: (562, 49).
- cidade 5: (576, 253).

Para criar um conjunto de novas cidades com posições aleatórias, usamos o código:

```
cities_locations = [(random.randint(0, 100), random.randint(0, 100)) for _ in range(N_CITIES)]
```

Esse código cria um número de N_CITIES cidades em posições aleatórias, com valores entre 0 e 100 nos eixos x e y.

**Codificação**

Em algoritmos genéticos, um indivíduo deve sempre representar uma solução viável para o problema. Portanto, uma solução nesse caso precisa descrever o caminho, ou seja, a ordem em que os pontos são visitados, sem repetir os pontos e visitando todos os pontos.

Logo, uma possível solução pode ser representada por um vetor contendo a localização das cidades na ordem em que são visitadas, por exemplo:

A solução [(576, 253), (733, 251), (562, 49), (706, 87), (546, 97)], representa as cidades sendo visitadas na ordem em que aparecem no vetor, produzindo o caminho abaixo:

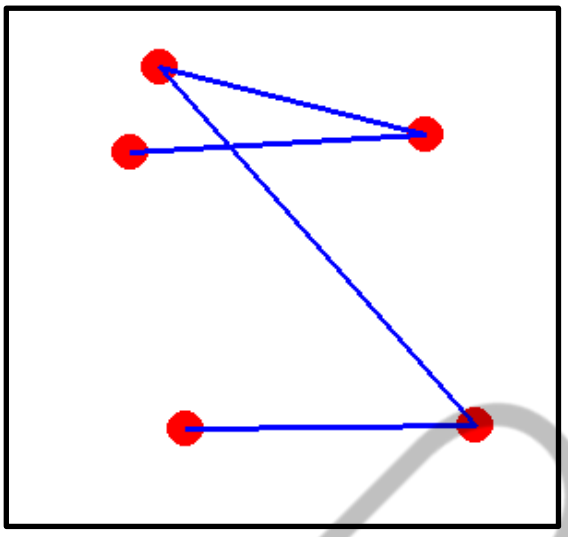

Figura 2 - Possível solução para o problema
Fonte: Elaborado pelo autor (2024), adaptado por FIAP (2024)

### Criação Da Geração Inicial

Agora que temos a codificação inicial, precisamos criar uma geração inicial, que basicamente consiste em variações dos caminhos. O código para fazer a criação da população inicial é descrita abaixo:

```
def generate_random_population(cities_location: List[Tuple[float, float]], population_size: int) -> List[List[Tuple[float, float]]]:
    """
    Generate a random population of routes for a given set of cities.

    Parameters:
    - cities_location (List[Tuple[float, float]]): A list of tuples representing the locations of cities,
      where each tuple contains the latitude and longitude.
    - population_size (int): The size of the population, i.e., the number of routes to generate.

    Returns:
    List[List[Tuple[float, float]]]: A list of routes, where each route is represented as a list of city locations.
    """
    return [random.sample(cities_location, len(cities_location)) for _ in range(population_size)]
```

O código recebe o vetor com as posições da cidade gerada inicialmente e retorna um número de indivíduos dado pelo parâmetro **populaton_size**.

### Cálculo de Fitness

Agora que temos uma população, precisamos avaliar o quão bem cada indivíduo da população está performando. Para isso, vamos usar a função **calculate_fitness**. Essa função recebe um indivíduo, ou caminho, e retorna a distância percorrida, em pixels, para visitar todas as cidades.

```
def calculate_fitness(path: List[Tuple[float, float]]) -> float:
    """
    Calculate the fitness of a given path based on the total
Euclidean distance.

    Parameters:
    - path (List[Tuple[float, float]]): A list of tuples
representing the path,
      where each tuple contains the coordinates of a point.

    Returns:
    float: The total Euclidean distance of the path.
    """
    distance = 0
    n = len(path)
    for i in range(n):
        distance += calculate_distance(path[i], path[(i + 1) % n])

    return distance
```

Essa função utiliza outra função chamada **calculate_distance** que, dado dos pontos, retorna a distância euclidiana entre esses dois pontos.

```
def calculate_distance(point1: Tuple[float, float], point2:
Tuple[float, float]) -> float:
    """
    Calculate the Euclidean distance between two points.

    Parameters:
    - point1 (Tuple[float, float]): The coordinates of the first
point.
    - point2 (Tuple[float, float]): The coordinates of the second
point.

    Returns:
    float: The Euclidean distance between the two points.
    """
    return math.sqrt((point1[0] - point2[0]) ** 2 + (point1[1] -
point2[1]) ** 2)
```

Nós aplicamos a função **calculate_fitness** para todos os indivíduos da população.

```
population_fitness = [calculate_fitness(individual) for individual
in population]
```

**Seleção**

Agora, com os valores de fitness de toda a população em mãos, nós sabemos como cada indivíduo performou, em outras palavras, qual é a solução com menor distância disponível. Assim, podemos aplicar diferentes métodos para seleção, vamos usar o método de seleção aleatória, mas ponderado pelo valor de fitness.

```
probability = 1 / np.array(population_fitness)
parent1, parent2 = random.choices(population, weights=probability,
k=2)
```

A probabilidade de um indivíduo ser selecionado é o inverso de sua distância, pois quanto menor a distância, maior queremos que seja a probabilidade.

Pronto, assim temos os indivíduos selecionados prontos para passarem pelo cruzamento para criar a população, que poderá ou não sofrer mutação.

**Cálculo de Fitness**

Como vimos, o cálculo de fitness é feito usando a função **calc_distance,** que retorna a distância euclidiana dado dois pontos. Entretanto, esse método é ineficiente computacionalmente pois executa repetidamente os mesmos cálculos de distância entre as cidades.

Uma abordagem mais eficiente é usando a matriz adjacente, ou matriz de distâncias. No contexto do “Problema do Caixeiro Viajante” (PCV), a matriz adjacente é uma matriz quadrada que representa as distâncias entre todas as cidades no problema. Se houver $n$ cidades, a matriz terá dimensões $n \times n$, e a entrada $D[i][j]$ na linha $i$-ésima e coluna $j$-ésima representará a distância entre a cidade i e a cidade j. A matriz é chamada de "adjacente" porque mostra a adjacência ou conectividade entre as cidades.

A distância total de uma rota é encontrada somando as distâncias entre cada par consecutivo de cidades na rota, conforme representado pelos índices na matriz adjacente. Se a rota for representada por uma lista de cidades visitadas $(R = [c_1, c_2, \ldots, c_n])$, onde $(c_i)$ é o índice da cidade na matriz, então a distância total $(L(R))$ ao longo dessa rota é dada por:

$$L(R) = D[c_1][c_2] + D[c_2][c_3] + \ldots + D[c_{n-1}][c_n] + D[c_n][c_1]$$

Essa fórmula reflete a ideia de que a rota deve começar e terminar na mesma cidade. Se não for necessário voltar à cidade de origem, a última parcela $(D[c_n][c_1])$ pode ser omitida.

**Exemplo:** aqui está um exemplo para ilustrar como a matriz adjacente é usada para calcular a distância total de uma rota:

Suponha que temos as seguintes distâncias entre quatro cidades:

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{bmatrix}$$

Esta é a matriz adjacente. Agora, se tivermos uma rota $(R = [0, 2, 1, 3])$, isso significa que estamos visitando as cidades na ordem 0 (primeira cidade), 2, 1 e 3. A distância total $(L(R))$ ao longo dessa rota é calculada da seguinte forma:

$$L(R) = D[0][2] + D[2][1] + D[1][3] + D[3][0]$$

Substituindo os valores da matriz adjacente, obtemos:

$$L(R) = 15 + 35 + 25 + 20 = 95$$

Portanto, a distância total ao longo da rota $(R)$ é de 95 unidades. Isso ilustra como a matriz adjacente é usada para calcular eficientemente a distância total de uma rota no Problema do Caixeiro Viajante.

**Eficiência Computacional**

Usar uma matriz adjacente no Problema do Caixeiro Viajante é mais rápido computacionalmente porque permite acesso direto às distâncias pré-calculadas entre todas as cidades, eliminando a necessidade de recalcular as distâncias a cada avaliação do fitness. Enquanto a distância euclidiana exigiria operações matemáticas complexas para cada par de cidades, a matriz oferece uma abordagem eficiente, com acesso constante aos valores de distância. Isso reduz significativamente a carga computacional, resultando em um desempenho mais rápido, especialmente em problemas com um grande número de cidades, onde a complexidade computacional das operações euclidianas aumentaria exponencialmente. O uso da matriz adjacente simplifica o processo, tornando-o mais eficiente e adequado para a aplicação em algoritmos genéticos e métodos de otimização no contexto do PCV.

Vá além, e modifique o código para o cálculo de fitness que fizemos em aula, que calcula a distância euclidiana em cada interação, e substitua por uma matriz de distâncias, que é calculada no início do código e depois apenas consultada. Meça o tempo médio de execução da função do cálculo de Fitness antes e depois dessa modificação e veja a melhora que obteve.

**Hotstart - Geração Inicial**

A abordagem inicial da nossa geração é bastante simples: criamos uma população de indivíduos aleatórios, bagunçando a ordem das cidades para formar novos conjuntos. Como resultado, surgem soluções iniciais de maneira espontânea. Ao executar o algoritmo algumas vezes, é notável que, em certos casos, encontramos soluções muito eficazes logo de início, acelerando significativamente o processo de convergência para uma solução satisfatória.

No entanto, em algumas execuções, a sorte pode não estar do nosso lado, e um indivíduo inicial de alta qualidade pode não ser gerado aleatoriamente, especialmente em problemas com um grande número de cidades.

Uma estratégia para aprimorar a população inicial é criar indivíduos com base em heurísticas, em vez de depender apenas da aleatoriedade. Embora métodos heurísticos possam demandar um tempo ligeiramente maior no início da execução do algoritmo genético em comparação com a geração aleatória, essa abordagem pode economizar considerável tempo ao introduzir desde o princípio soluções de qualidade na população. Essa técnica de incorporar deliberadamente um indivíduo potencialmente eficiente é conhecida como "hotstart".

Você pode ir além e introduzir uma ou heurística para o **hotstart** da população. Para isso, use heurísticas determinísticas para criar soluções iniciais. Uma heurística determinística é um método que fornece soluções aproximadas para um problema, sem garantia de encontrar a solução ótima. Diferente de abordagens mais rigorosas, como algoritmos exatos, as heurísticas determinísticas são rápidas e eficientes, mas não asseguram a solução ideal.

Precisamente, a característica determinística dessas heurísticas é fundamental para sua aplicação no hotstart da população. Ao contrário de algoritmos genéticos que podem iterar indefinidamente por gerações, as heurísticas determinísticas possuem um método com início e fim definidos. Sua natureza determinística garante que, ao serem aplicadas a um mesmo problema, sempre conduzirão ao mesmo resultado, proporcionando consistência nos resultados obtidos. Essa previsibilidade é valiosa, uma vez que contribui para a estabilidade e reprodutibilidade do processo, tornando-as particularmente adequadas para a geração inicial de soluções no contexto do Problema do Caixeiro Viajante.

Aqui estão algumas heurísticas determinísticas de como funcionam para resolver o Problema do Caixeiro Viajante.

**Vizinho Mais Próximo (Nearest Neighbor):** começa em uma cidade qualquer e, em cada passo, seleciona a cidade mais próxima ainda não visitada como próxima parada.

**Envoltória Convexa (Convex Hull):** identifica o conjunto convexo das cidades, formando um polígono convexo que abrange todas as cidades. O percurso do caixeiro é então determinado ao seguir a fronteira desse polígono.

**Inserção Mais Barata (Cheapest Insertion):** inicia com um subconjunto de cidades e, a cada passo, insere a cidade não incluída que tem o menor custo adicional ao percurso existente.

**Mínima Aresta de Vértice (Minimum Spanning Tree):** conecta inicialmente os dois vértices mais próximos e, em seguida, adiciona as arestas mais curtas conectando vértices já incluídos ao percurso.

**Árvore Geradora Mínima (Minimum Spanning Tree - MST):** constrói uma árvore geradora mínima do grafo completo, utilizando algoritmos como Prim ou Kruskal, e em seguida, realiza um passeio pré-ordem na árvore para obter o percurso do caixeiro.

Estas heurísticas determinísticas oferecem abordagens simplificadas e eficientes para o Problema do Caixeiro Viajante, embora não garantam a solução ótima em todos os casos.

## O QUE VOCÊ VIU NESTA AULA?

Na Aula 4, mergulhamos na representação de Indivíduos, Aptidão e Seleção, explorando de maneira prática a solução do problema do Problema do Caixeiro Viajante em Python, utilizando a biblioteca Pygame e a IDE Spyder. Discutimos a essência de como um indivíduo pode representar uma solução, sendo este um simples aglomerado de cidades. Na abordagem da função de aptidão, calculamos distâncias euclidianas, e elucidamos como esse cálculo pode ser otimizado através do uso de uma tabela de distâncias.

Ao adotar essa estratégia, trocamos o desafio computacional por um custo de memória em instâncias mais volumosas do problema. Na etapa de seleção, exploramos a escolha aleatória dos 10 primeiros indivíduos, eliminando a possibilidade de estagnação ao nunca selecionar soluções inferiores. Além disso, introduzimos a seleção probabilística utilizando a função random.choices, ampliando as possibilidades e promovendo maior variabilidade na busca pela solução ideal.

Gostaríamos de saber sua opinião sobre o conteúdo! Compartilhe seus comentários e dúvidas no Discord, onde estamos disponíveis na comunidade para responder a perguntas, promover networking, fornecer avisos e muito mais. Junte-se a nós!

# REFERÊNCIAS

CHENEY, N.; MACCURDY, R.; CLUNE, J.; LIPSON, H. **Unshackling Evolution: Envolving Soft Robots with Multiple Materials and a Powerful Generative Encoding**. Cornell University. [s.d]. Disponível em: http://jeffclune.com/publications/2013_Softbots_GECCO.pdf. Acesso em: 25 mar. 2024.

POLIMANTE, S.; PRATI, R.; KLEINSCHMIDT, J.H. **Evolução multiobjetivo de trajetórias como múltiplas curvas de Bézier para VANTs**. 2020. [s.l.]. Disponível em: https://www.researchgate.net/publication/376134695_Evolucao_multiobjetivo_de_trajetorias_como_multiplas_curvas_de_Bezier_para_VANTs. Acesso em: 25 mar. 2024.

POLIMANTE, S.; PRATI, R.; KLEINSCHMIDT, J.H. **Otimização Multiobjetivo de Trajetórias de VANTs Utilizando Curvas de Bézier e Algoritmos Genéticos**. Conference XIV Brazilian Congress of Computational Intelligence, Belém, 2019. Disponível em: https://www.researchgate.net/publication/337051141_Otimizacao_Multiobjetivo_de_Trajetorias_de_VANTs_Utilizando_Curvas_de_Bezier_e_Algoritmos_Geneticos. Acesso em: 25 mar. 2024.

STANLEY, O. K.; MIIKULAINEN, R.; et.al. **Envolving Neural Networks through Augmenting Topologies.** IEEE Explore – MIT Press. 2002. Disponível em: https://ieeexplore.ieee.org/document/6790655. Acesso em: 25 mar. 2024.

SUN, Y.; XUE, B.; ZHANG, M. YEN, G.G. **Envolving Deep Convolutional Neural Networks for Image Classification**. Cornell University. 2017. Disponível em: https://arxiv.org/abs/1710.10741. Acesso em: 25 mar. 2024.

TANGHE, K. B. **On the origin of species: the story of Darwin's title**. The Royal Society Journal of the History of Science. 2018. Disponível em: https://royalsocietypublishing.org/doi/10.1098/rsnr.2018.0015. Acesso em: 25 mar. 2024.

## PALAVRAS-CHAVE

Genetic Algorithms. Fitness Function. Selection.