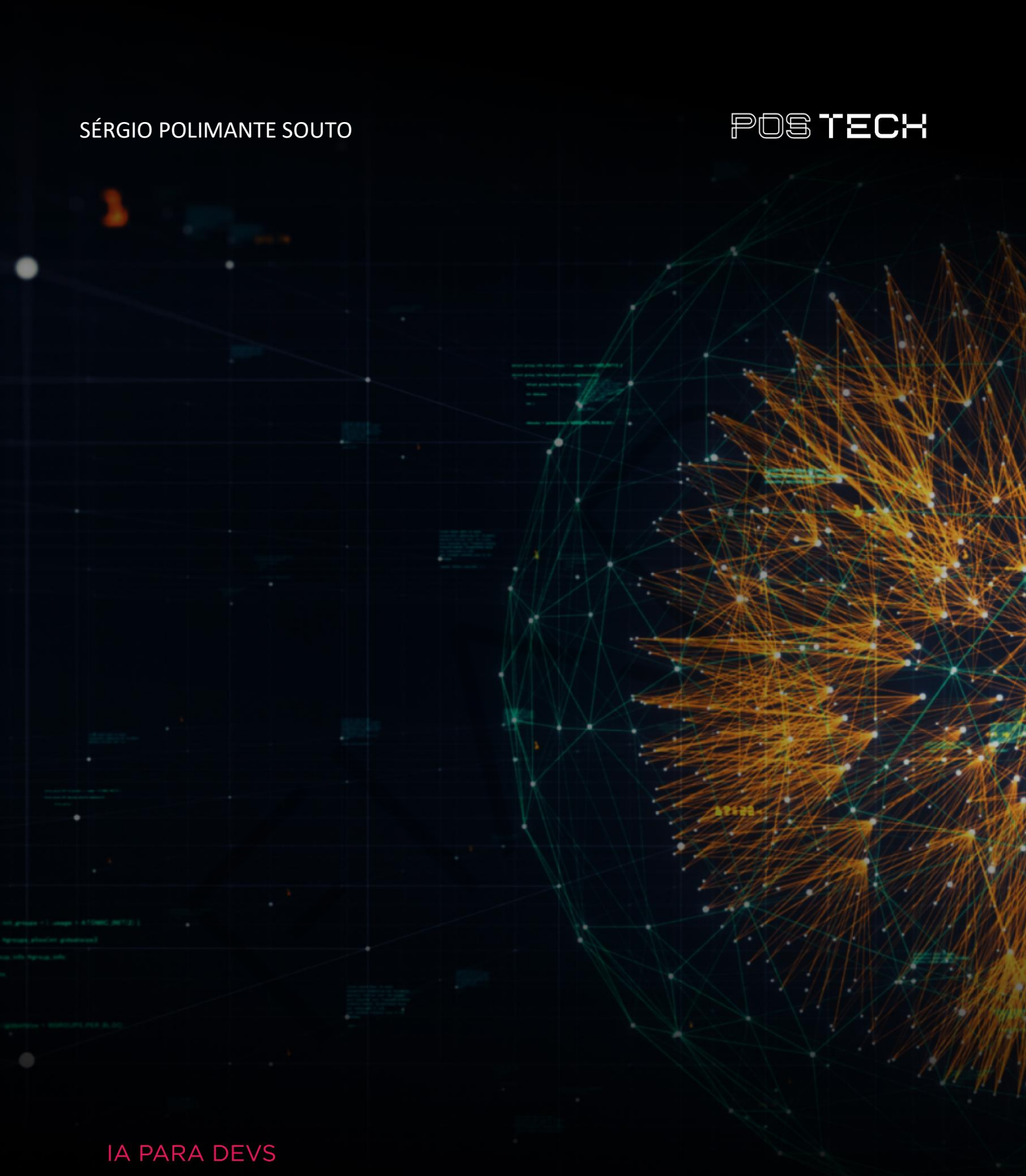
SÉRGIO POLIMANTE SOUTO



IA PARA DEVS

INTRODUÇÃO AO ALGORITMO GENÉTICO

AULA 05

SUMÁRIO

O QUE VEM POR AÍ?

Nesta aula, continuaremos nossa exploração prática criando os operadores genéticos de cruzamento e mutação. Além disso, trabalharemos na visualização dos resultados utilizando gráficos e uma interface gráfica para visualizar a melhor rota em tempo real usando Pygame.

Abordaremos o crossover, com destaque para o método Ordered (ox) aplicado ao “Problema do Caixeiro Viajante”, mostrando como os filhos são gerados. Discutiremos a customização da função de crossover, ressaltando sua dependência do indivíduo. Em seguida, exploraremos a mutação, focando no método Swap. Além disso, examinaremos a condição de término de um algoritmo genético.

Utilizaremos o Pygame para criar visualizações interativas dos resultados. Apresentaremos o Pygame, suas funções de criação de tela, controle de frames e as funções para desenhar as cidades e a melhor trajetória na tela.

Prepare-se para uma aula prática e dinâmica, explorando os detalhes dos operadores genéticos e visualizando os resultados em tempo real.

HANDS ON

Vamos dar continuidade a aula anterior criando nossos operadores genéticos de cruzamento e mutação. Além disso, vamos ver também as condições de término do algoritmo genético e as funções necessárias para criar a visualização interativa e em tempo real usando Pygame. Com isso, teremos visto todas as etapas de uma solução implementada com algoritmo genético.

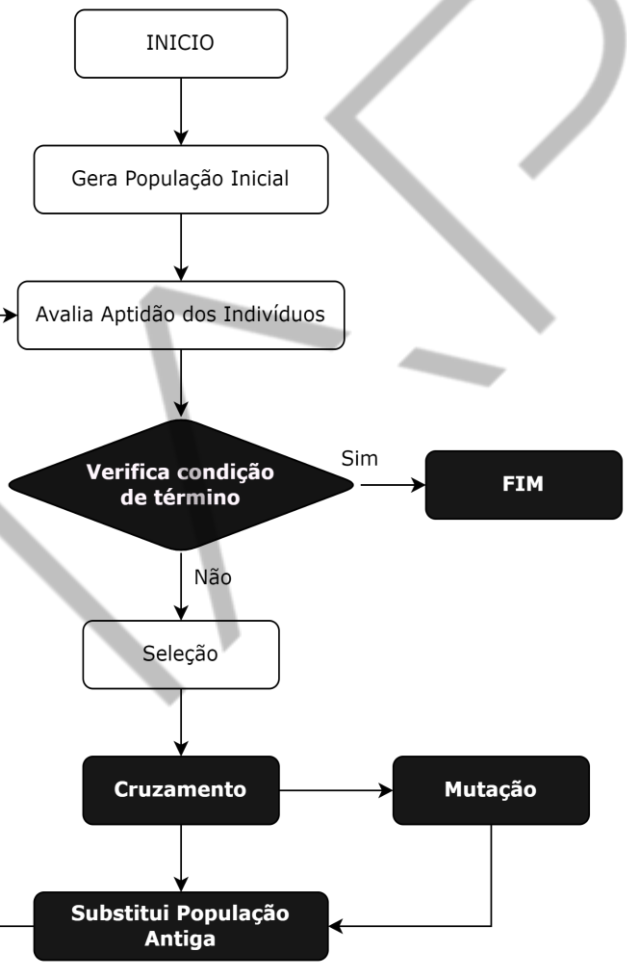


Figura 1 - Fluxograma genérico do algoritmo genético
Fonte: elaborado pelo autor (2024), adaptado por FIAP (2024)

Na aula anterior, nós vimos os métodos de seleção, que seleciona os indivíduos da população com base em seus valores de aptidão. Os indivíduos selecionados irão cruzar para produzir a nova população. Para isso, precisamos do operador genético de Cruzamento.

SAIBA MAIS

CRUZAMENTO

A função de cruzamento desempenha um papel crucial em algoritmos genéticos, especialmente quando se trata de resolver problemas complexos, como o “Problema do Caixeiro Viajante”.

Em termos simples as funções de cruzamento devem receber dois indivíduos da população atual, misturar seu material genético e produzir uma nova solução contendo características das duas soluções anteriores.

É fundamental que o método de cruzamento de produza soluções válidas para o problema. Em problemas combinatórios, esses métodos não são tão simples quanto em problemas binários ou reais, pois as soluções possuem restrições inerentes ao problema. Por exemplo, no “Problema do Caixeiro Viajante”, as soluções precisam visitar todas as cidades, não tendo repetições e nem falta de nenhuma cidade.

Se aplicássemos o método de cruzamento **single-point** crossover, por exemplo, produziremos muitas soluções inválidas. Por exemplo, considere os indivíduos que visitem as sete cidades, de A a G.

```
P0 = [A, B, C, D, E, F, G]

P1 = [B, C, E, A, F, G, D]
```

Suponha que vamos usar o método de single-point com corte no índice 3.

```
id = [0, 1, 2, | 3, 4, 5, 6]

P0 = [A, B, C, | D, E, F, G]

P1 = [B, C, E, | A, F, G, D]
```

Vamos fazer o corte no ponto de índice (id) igual a 3 e pegar cada parte para um novo indivíduo filho:

```
F0 = [A, B, C, A, F, G, D]

F1 = [B, C, E, D, E, F, G]
```

Note que ambos os filhos produzidos são soluções inválidas. O indivíduo F0, por exemplo, visita duas vezes a cidade A e não visita a cidade E. Similarmente, o indivíduo F1 não visita a cidade A e visita duas vezes a cidade E.

Portanto, precisamos usar funções de crossover que criem indivíduos válidos a partir dos pais. Existem diversos tipos de cruzamento para problemas combinatórios como o do caixeiro viajante e que produzem soluções validas, como Order Crossover (ox), Partially Mapped Crossover (pmx), Cycle Crossover (cx), Edge Recombination Crossover (erx). Para nosso código, vamos usar a função chamada Ordered Crossover (OX1).

ORDERED CROSSOVER (OX1)

O crossover ordenado OX1 (Order Crossover 1) é uma estratégia fundamental em algoritmos genéticos para problemas combinatórios. Neste método, uma subsequência contínua de genes é selecionada aleatoriamente de um dos pais e copiada diretamente para o descendente. Em seguida, a ordem dos genes restantes no descendente é determinada pela preservação da ordem relativa dos genes no segundo pai, sem duplicatas.

O OX1 garante a produção de soluções válidas, uma vez que os descendentes mantêm a estrutura ordenada dos pais. Essa abordagem é particularmente útil no TSP, onde é crucial preservar a sequência de cidades a serem visitadas, contribuindo para a eficiência do algoritmo genético na busca por soluções de alta qualidade ao longo das gerações.

Vamos relembrar um exemplo prático sobre como ela funciona:

Dados os dois pais selecionados:

P0 = (A, B, C, D, E, F, G, H, I, J)
P1 = (B, D, A, H, J, C, E, G, F, I)

Selecione uma substring, por exemplo, entre 2 e 7 e inicialize os filhos com os genes de cada pai

F1 = (_, _, C, D, E, F, G, _, _, _)
F2 = (_, _, A, H, J, C, E, _, _, _)

Use os genes do outro pai para completar o gene dos filhos na ordem com que os genes aparecem no pai.

F1 = (B, A, C, D, E, F, G, H, J, I)
F2 = (B, D, A, H, J, C, E, F, G, I)

Os novos filhos preservam parcialmente a ordem dos pais e introduzem nova combinações.

A implementação é feita usando a função order_crossover, abaixo:

```
def order_crossover(parent1: List[Tuple[float, float]], parent2:
List[Tuple[float, float]]) -> List[Tuple[float, float]]:
    """
    Perform order crossover (OX) between two parent sequences to
create a child sequence.

    Parameters:
    - parent1 (List[Tuple[float, float]]): The first parent
sequence.
    - parent2 (List[Tuple[float, float]]): The second parent
sequence.

    Returns:
    List[Tuple[float, float]]: The child sequence resulting from
the order crossover.
    """
    length = len(parent1)

    # Choose two random indices for the crossover
    start_index = random.randint(0, length - 1)
    end_index = random.randint(start_index + 1, length)

    # Initialize the child with a copy of the substring from
```

```
parent1
    child = parent1[start_index:end_index]

    # Fill in the remaining positions with genes from parent2
    remaining_positions = [i for i in range(length) if i <
start_index or i >= end_index]
    remaining_genes = [gene for gene in parent2 if gene not in
child]

    for position, gene in zip(remaining_positions,
remaining_genes):
        child.insert(position, gene)

    return child
```

Agora, com um novo indivíduo produzido pela função de crossover, vamos aplicar o operador genético de mutação.

MUTAÇÃO

O operador genético de mutação em algoritmos genéticos desempenha um papel crucial na introdução de diversidade nas populações de soluções ao longo das gerações. Sua função principal é realizar pequenas modificações aleatórias nos genes das soluções, contribuindo para a exploração de novas regiões do espaço de busca. Ao introduzir alterações nos indivíduos, a mutação ajuda a evitar a convergência prematura para soluções subótimas, promovendo a descoberta de soluções mais diversas e potencialmente melhores. No contexto do “Problema do Caixeiro Viajante”, por exemplo, diferentes técnicas de mutação, como trocas, inversões ou deslocamentos de cidades no percurso, possibilitam a criação de novas rotas e a busca por soluções mais eficientes. A taxa de mutação é um parâmetro ajustável que equilibra a necessidade de explorar novas soluções com a estabilidade das soluções já encontradas ao longo das iterações do algoritmo genético.

Existem diferentes tipos de mutação para um problema do tipo combinatório como o do caixeiro viajante como mutação de troca (swap mutation), mutação de inversão (inversion mutation), mutação de deslocamento (displacement mutation), mutação de inserção (insertion mutation), mutação por scramble (scramble mutation).

Vamos seguir com o tipo mais simples, swamp mutation, que é implementada usando o algoritmo abaixo:

```
def mutate(solution:  List[Tuple[float, float]],
mutation_probability: float) ->  List[Tuple[float, float]]:
    """
    Mutate a solution by inverting a segment of the sequence with a
given mutation probability.

    Parameters:
    - solution (List[int]): The solution sequence to be mutated.
    - mutation_probability (float): The probability of mutation for
each individual in the solution.

    Returns:
    List[int]: The mutated solution sequence.
    """
    mutated_solution = copy.deepcopy(solution)

    # Check if mutation should occur
    if random.random() < mutation_probability:

        # Ensure there are at least two cities to perform a swap
        if len(solution) < 2:
            return solution

        # Select a random index (excluding the last index) for
swapping
        index = random.randint(0, len(solution) - 2)

        # Swap the cities at the selected index and the next index
        mutated_solution[index], mutated_solution[index + 1] =
solution[index + 1], solution[index]

    return mutated_solution
```

TESTE DOS OPERADORES GENÉTICOS

Com isso, finalizamos os operadores genéticos. Agora que temos todas as funções para o algoritmo genético, vamos escrever um código que constrói o fluxo de execução do algoritmo genético e testar se está funcionando:

```
if __name__ == '__main__':
    N_CITIES = 10

    POPULATION_SIZE = 100
    N_GENERATIONS = 100
    MUTATION_PROBABILITY = 0.3
    cities_locations = [(random.randint(0, 100), random.randint(0,
100))
                for _ in range(N_CITIES)]

    # CREATE INITIAL POPULATION
    population = generate_random_population(cities_locations,
POPULATION_SIZE)

    # Lists to store best fitness and generation for plotting
    best_fitness_values = []
    best_solutions = []

    for generation in range(N_GENERATIONS):


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

        population, population_fitness =
sort_population(population,  population_fitness)

        best_fitness = calculate_fitness(population[0])
        best_solution = population[0]

        best_fitness_values.append(best_fitness)
        best_solutions.append(best_solution)

        print(f"Generation {generation}: Best fitness =
{best_fitness}")

        new_population = [population[0]]  # Keep the best
individual: ELITISM

        while len(new_population) < POPULATION_SIZE:

            # SELECTION
```

```
            parent1, parent2 = random.choices(population[:10], k=2)
# Select parents from the top 10 individuals

            # CROSSOVER
            child1 = order_crossover(parent1, parent2)

            ## MUTATION
            child1 = mutate(child1, MUTATION_PROBABILITY)

            new_population.append(child1)


        print('generation: ', generation)
        population = new_population
```

O código acima nos permite visualizar o funcionamento do nosso algoritmo ao longo das gerações. Podemos controlar o código usando os parâmetros globais:

- N_CITIES: número de cidades criadas com posições aleatórias;
- POPULATION_SIZE: tamanho da população do algoritmo genético;
- N_GENERATIONS: número de gerações para ser rodada na execução do algoritmo;
- MUTATION_PROBABILITY: probabilidade de ocorrer uma mutação no novo indivíduo criado.

Podemos observar uma diminuição no fitness impresso no console ao longo das gerações, o que indica que nosso algoritmo está funcionando.

VISUALIZAÇÃO USANDO PYGAME

Vamos criar uma interface gráfica usando Pygame para visualizar a evolução das gerações do algoritmo genético em tempo real. Isso vai nos dar um melhor entendimento sobre como o algoritmo busca trajetórias mais curtas.

Primeiro, vamos começar vendo os recursos que precisamos implementar para construir a tela do Pygame.

Inicialmente, vamos definir as seguintes variáveis globais:

```
# Define constant values
# pygame
WIDTH, HEIGHT = 800, 400
NODE_RADIUS = 10
FPS = 30
PLOT_X_OFFSET = 450
```

`WIDTH, HEIGHT = 800, 400:` define as constantes WIDTH e HEIGHT com os valores 800 e 400, representando as dimensões da tela.

`NODE_RADIUS = 10:` define a constante NODE_RADIUS com o valor 10, que representa o raio dos nós ou pontos na visualização.

`FPS = 30:` define a constante FPS com o valor 30, indicando a taxa desejada de quadros por segundo (frames per second) para a atualização da tela.

`PLOT_X_OFFSET = 450:` define a constante PLOT_X_OFFSET com o valor 450, representando o deslocamento horizontal utilizado na visualização ou gráfico. Esse deslocamento pode ser útil para ajustar a posição dos elementos na tela.

```
# Initialize Pygame
pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("TSP Solver using Pygame")
clock = pygame.time.Clock()
```

`pygame.init():` inicializa o módulo Pygame para preparar o ambiente de desenvolvimento.

`screen = pygame.display.set_mode((WIDTH, HEIGHT)):` cria uma tela com as dimensões especificadas por WIDTH e HEIGHT para renderizar gráficos e interações.

`pygame.display.set_caption("TSP Solver using Pygame"):` define o título da janela da aplicação como "TSP Solver using Pygame".

`clock = pygame.time.Clock():` cria um objeto “Clock” para controlar a taxa de atualização da tela e garantir uma experiência de usuário mais suave.

Em seguida, vamos criar o controle do loop principal:

```
# Main game loop
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_q:
                running = False
```

O código acima define as duas condições de saída do loop:

- `pygame.QUIT`: quando a tela for fechada.
- `event.key == pygame.K_q:` quando a tecla **q** for pressionada.

As últimas funções de controle da execução do Pygame são as que encerram a visualização, esses são os últimos comandos a serem executados:

```
# exit software
pygame.quit()
sys.exit()
```

pygame.quit(): encerra todos os módulos do Pygame e libera os recursos utilizados.

sys.exit(): encerra o interpretador Python, finalizando completamente a execução do programa. É necessário importar o módulo sys para usar esta função.

Agora, com as rotinas de criação e controle do Pygame implementados, vamos criar as funções que de fato vão desenhar o que queremos ver na tela (screen) do Pygame, que serão:

- Desenhar gráfico de fitness.
- Desenhar cidades.
- Desenhar melhor trajetória

Desenhando Gráfico de Fitness

Antes de partirmos para desenhar as cidades e as trajetórias, vamos desenhar um gráfico para visualizar a evolução das gerações. Esse gráfico vai mostrar o valor de aptidão, que no caso do caixeiro viajante é a distância total da trajetória, do melhor indivíduo de cada geração. Então no eixo x temos as gerações e no eixo y a distância da melhor solução naquela geração.

Para isso, vamos criar a função **draw_plot** no arquivo draw_functions.py:

```
def draw_plot(screen: pygame.Surface, x: list, y: list, x_label:
str = 'Generation', y_label: str = 'Fitness') -> None:
    """
    Draw a plot on a Pygame screen using Matplotlib.

    Parameters:
    - screen (pygame.Surface): The Pygame surface to draw the plot
on.
    - x (list): The x-axis values.
    - y (list): The y-axis values.
    - x_label (str): Label for the x-axis (default is
'Generation').
    - y_label (str): Label for the y-axis (default is 'Fitness').
    """
    fig, ax = plt.subplots(figsize=(4, 4), dpi=100)
    ax.plot(x, y)
    ax.set_ylabel(y_label)
    ax.set_xlabel(x_label)
    plt.tight_layout()

    canvas = FigureCanvasAgg(fig)
    canvas.draw()
    renderer = canvas.get_renderer()
    raw_data = renderer.tostring_rgb()

    size = canvas.get_width_height()
```

```
    surf = pygame.image.fromstring(raw_data, size, "RGB")
    screen.blit(surf, (0, 0))
```

Essa função recebe os seguinte parâmetros:

- screen: a tela do pygame em que será desenhado (screen);
- x: valores do eixo x;
- y: valores do eixo y;
- x_label: texto do eixo x;
- y_label: texto do eixo y.

Adicionamos a linha abaixo em nosso loop principal para desenhar o plot:

```
    draw_plot(screen, list(range(len(best_fitness_values))),
best_fitness_values, y_label = "Fitness - Distance (pxls)")
```

Desenhando Cidades

As cidades podem ser representadas de maneira simples por círculos vermelhos na tela. Para desenhar as cidades, usaremos a função **draw_cities** no arquivo draw_functions.py:

```
def draw_cities(screen: pygame.Surface, cities_locations:
List[Tuple[int, int]], rgb_color: Tuple[int, int, int],
node_radius: int) -> None:
    """
    Draws circles representing cities on the given Pygame screen.

    Parameters:
    - screen (pygame.Surface): The Pygame surface on which to draw
the cities.
    - cities_locations (List[Tuple[int, int]]): List of (x, y)
coordinates representing the locations of cities.
    - rgb_color (Tuple[int, int, int]): Tuple of three integers (R,
G, B) representing the color of the city circles.
    - node_radius (int): The radius of the city circles.
```

```
    Returns:
    None
    """
    for city_location in cities_locations:
        pygame.draw.circle(screen, rgb_color, city_location,
node_radius)
```

Essa função recebe os seguintes parâmetros:

- screen: a tela do Pygame em que será desenhado (screen).
- cities_locaton: lista com tuplas de valores x e y da posição das cidades.
- rgb_color: cor RGB (0, 0, 0) de cor das cidades.
- node_radius: raio do círculo em pixels.

Adicionamos a linha abaixo em nosso loop principal para desenhar as cidades:

```
    draw_cities(screen, cities_locations, RED, NODE_RADIUS)
```

Desenhando o Caminho

Finalmente, após ter as cidades desenhadas, vamos criar a função que desenha as cidades. Para isso, vamos criar a função **draw_paths** no arquivo draw_functions.py:

```
def draw_paths(screen: pygame.Surface, path: List[Tuple[int, int]],
rgb_color: Tuple[int, int, int], width: int = 1):
    """
    Draw a path on a Pygame screen.

    Parameters:
    - screen (pygame.Surface): The Pygame surface to draw the path
on.
    - path (List[Tuple[int, int]]): List of tuples representing the
coordinates of the path.
    - rgb_color (Tuple[int, int, int]): RGB values for the color of
the path.
    - width (int): Width of the path lines (default is 1).
```

```
    """
    pygame.draw.lines(screen, rgb_color, True, path, width=width)
```

Essa função recebe os seguintes parâmetros:

- screen: a tela do Pygame em que será desenhado (screen).
- path: lista com tuplas de valores x e y da posição das cidades na ordem em que são visitadas.
- rgb_color: cor RGB (0, 0, 0) de cor das cidades.
- width: espessura da linha criada.

Adicionamos a linha abaixo em nosso loop principal para desenhar o melhor caminho:

```
    draw_paths(screen, best_solution,BLUE, width=3)
```

Além disso, vamos adicionar um caminho extra, sendo o segundo melhor, para termos apenas uma visão mais interativa de que o algoritmo está fazendo a busca. Nesse segundo caminho, vamos usar um tom mais discreto, cinza, com uma espessura menor.

```
    draw_path(screen, population[1], rgb_color=(128, 128, 128),
width=1)
```

Testando o Algoritmo

Pronto! Agora temos tudo que precisamos para rodar nosso algoritmo genético e vê-lo funcionando em tempo real.

Inicialização do Problema

Temos três (3) maneiras de inicializar o posicionamento das cidades:

- **Criação de cidades em ordem aleatória**
    - Esta inicialização criará N_CITIES cidades em posições aleatórias respeitando o tamanho do nosso frame. As cidades são criadas usando o código presente no link:

https://github.com/sergiopolimante/genetic_algorithm_tsp/blob/main/tsp.py.

- Utilização das instâncias “presetadas”
  - Essas são instâncias “presetadas” para podermos comparar a execução do nosso algoritmo genético com um mesmo conjunto de cidades.
  - Use essas cidades para testar modificações no seu algoritmo. Implemente hotstart, diferentes técnicas de crossover e mutação, autoajuste dinâmico de parâmetros e outras técnicas, e veja como seu algoritmo modificado performa nesses problemas “presetados” conhecidos. Meça o número de gerações para chegar em uma boa solução otimizada.
  - Compartilhe com seus colegas no Discord, qual foi o menor valor que você encontrou para cada um dos problemas “presetados”.
- Utilização do benchmark att48
  - Esse é um problema benchmark com 48 cidades (capitais dos estados dos EUA). Este é um problema maior do que os problemas que vimos anteriormente (com 48 cidades), e fornece um maior nível de complexidade.
  - Diferente dos problemas “presetados”, neste nós conhecemos o caminho ótimo: a lista chamada att_48_cities_order que está no arquivo benchmark_att48.py.
  - Implemente hotstart, diferentes técnicas de crossover e mutação, autoajuste dinâmico de parâmetros, e outras técnicas e veja como seu algoritmo modificado performa neste problema complexo.
  - Compartilhe com seus colegas no Discord se você conseguiu chegar na resposta ótima. Quantas gerações você levou?

Pronto! Agora temos todos os elementos necessários para resolver o problema do caixeiro viajante usando algoritmo genético com uma interface gráfica.

Crossover

Existem diversos outros tipos de crossover para o problema do caixeiro viajante, tais como os listados abaixo. Vá além e implemente outras funções de crossover.

- **Order Crossover (OX):** seleciona um subconjunto de genes de um dos pais e preenche as posições restantes com a ordem dos genes do outro pai. Esse foi o que implementamos em nosso código.
- **Partially Mapped Crossover (PMX):** escolhe um subconjunto de genes de um dos pais e mapeia as posições correspondentes no outro pai para criar um filho. Conflitos são resolvidos trocando valores.
- **Cycle Crossover (CX):** identifica ciclos de genes entre dois pais e seleciona alternadamente ciclos para formar um filho.
- **Edge Recombination Crossover (ERX):** constrói um filho selecionando arestas com base nas arestas comuns dos pais. O objetivo é preservar a conectividade na rota resultante.
- **Position-Based Crossover (PBX):** seleciona um subconjunto de posições de um dos pais e preenche as posições restantes com genes do outro pai, mantendo a ordem de aparecimento.
- **Uniform Order-Based Crossover (UOX):** mistura a ordem dos genes entre dois pais com uma certa probabilidade, criando um filho.
- **Cycle Edge Crossover (CEX):** combina elementos dos métodos de crossover de ciclo e de aresta.
- **Greedy Crossover (GX):** constrói um filho selecionando a próxima cidade com base na aresta restante mais curta, favorecendo caminhos com distâncias menores.
- **Sequential Constructive Crossover (SCX):** constrói um filho selecionando iterativamente a próxima cidade com base na aresta restante mais curta.

Mutação

Também existem diferentes tipos de mutação, tais como as listadas abaixo. Algumas delas possuem maior intensidade de mutação (que é o quanto o indivíduo é

modificado), outras, menos. Você também pode criar um parâmetro de mutação chamado “mutation_intensity” para controlar o quão ‘forte’ será a mutação.

- **Swap Mutation:** realiza a troca de posição entre dois genes na sequência, alterando a ordem das cidades. Essa foi a que implementamos em nosso código.
- **Insertion Mutation:** remove uma cidade da sequência e a reinsere em uma posição diferente, alterando a ordem das cidades.
- **Inversion Mutation:** inverte a ordem de um segmento contínuo da sequência de cidades.
- **Scramble Mutation:** mistura aleatoriamente as posições de um subconjunto de cidades na sequência.
- **Displacement Mutation:** remove um segmento contínuo de cidades da sequência e reinsere esse segmento em uma posição diferente.
- **Inverted Displacement Mutation:** similar à Displacement Mutation, mas inverte a ordem do segmento antes de reinseri-lo.
- **Heuristic Mutation:** aplica uma heurística específica para melhorar a qualidade da solução, como a aplicação de um algoritmo de otimização local.
- **Random Resetting Mutation:** seleciona aleatoriamente uma posição na sequência e substitui a cidade atual por outra aleatória.
- **Shuffle Mutation:** embaralha aleatoriamente a ordem das cidades na sequência.
- **Segment Inversion Mutation:** inverte a ordem de um segmento arbitrário de cidades na sequência.

Benchmark

Benchmark é um padrão usado para comparar algoritmos e soluções. No campo da computação e otimização, ele fornece uma maneira justa e consistente de analisar e comparar o quão bem diferentes abordagens funcionam em condições controladas. No “Problema do Caixeiro Viajante” (TSP), (que busca encontrar a rota

mais eficiente para visitar todas as cidades uma vez), os benchmarks incluem conjuntos de testes com várias características, como o número e a distribuição de cidades. Usar benchmarks no TSP é crucial para avaliar a qualidade de algoritmos, permitindo comparações justas entre diferentes abordagens. Isso também ajuda a identificar algoritmos que se destacam em diferentes tipos de situações, proporcionando uma visão mais completa de seu desempenho.

Implementamos o benchmark att48 para avaliar algoritmos no “Problema do Caixeiro Viajante” (TSP). Esse benchmark consiste em 48 capitais dos EUA com distâncias conhecidas entre elas. A utilização do att48 permite uma avaliação direta e comparativa de diferentes abordagens para o TSP, proporcionando insights sobre a eficácia e eficiência dos algoritmos em um contexto prático e amplamente reconhecido. Essa implementação contribui para uma análise mais robusta e válida do desempenho dos algoritmos no TSP.

Para usar a instância do att48, use a inicialização com o código abaixo:

```
# Using att48 benchmark
WIDTH, HEIGHT = 1500, 800
att_cities_locations = np.array(att_48_cities_locations)
max_x = max(point[0] for point in att_cities_locations)
max_y = max(point[1] for point in att_cities_locations)
scale_x = (WIDTH - PLOT_X_OFFSET - NODE_RADIUS) / max_x
scale_y = HEIGHT/ max_y
cities_locations = [(int(point[0] * scale_x + PLOT_X_OFFSET),
int(point[1] * scale_y)) for point in att_cities_locations]
target_solution = [cities_locations[i-1] for i in
att_48_cities_order]
fitness_target_solution = calculate_fitness(target_solution )
print(f"Best Solution: {fitness_target_solution}")
```

Esse código carrega as posições das cidades e as “reescala” para que possam caber dentro da tela do Pygame. Mudar a escala (distância das cidades) não muda a complexidade do problema.

A distância da melhor solução, a solução ótima, é impressa no início do código. Ou seja, não existe nenhum outro caminho que seja menor do que esse.

Use essa informação para avaliar o quão perto sua solução foi capaz de chegar da solução ótima. Efetue modificações no algoritmo genético e veja qual o impacto que causam no desempenho do algoritmo.

Existem diversas outros benchmarks para o problema do caixeiro viajante. Alguns com número de cidades muito grande como The World TSP, com 1.904.711 de cidades.

O QUE VOCÊ VIU NESTA AULA?

Nesta aula, exploramos os Operadores Genéticos, com destaque para o Cruzamento (Crossover) e a Mutação, focando no método Swap. Abordamos o Ordered Crossover (ox) para o “Problema do Caixeiro Viajante” (tsp) e destacamos a customização do crossover, dependendo do indivíduo.

Discutimos a condição de término dos Algoritmos Genéticos para avaliar convergência. Utilizamos o Pygame para criar visualizações interativas, apresentando funções de tela, controle de frames e representação gráfica de traços e cidades. Em resumo, a aula proporcionou insights práticos sobre a aplicação dos operadores genéticos e sua visualização com Pygame.

Agora, gostaríamos de saber a sua opinião sobre o conteúdo. Compartilhe seus comentários e dúvidas no Discord, onde estamos disponíveis na comunidade para responder às 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 Operators. Crossover. Mutation. Ordered Crossover (OX).