Tech Challenge 2

Objetivo

O objetivo deste Tech Challenge, é projetar, implementar e testar um sistema que utiliza algoritmos genéticos para otimizar uma função ou resolver um problema complexo de otimização.

Problema

O problema a ser resolvido neste Tech Challenge, é um otimizador de rotas, projetamos e implementamos um algoritmo genético que encontrará o menor custo dentro uma origem e um destino.

Solução

O algoritmo desenvolvido, recebe uma lista de cidades e suas coordenadas(latitude e longitude), e calcula automaticamente o custo entre cada cidade baseado na distância entre as cidade, caso a distância entre uma cidade ou outra de uma diferença de 30, o custo é definido como infinito, com o objetivo para inviabilizar esta rota.

```python
def get_data():
    nodes = [
      "Los Angeles", "Atlanta", "Miami", "New York", "Chicago", "Seattle",
      "Vancouver", "Toronto", "Salt Lake City", "Denver", "Dallas", "Kansas City",
      "Memphis", "NashVille", "Columbus", "Calgary", "Regina", "Winnipeg"
    ]

    nodes_coordinates = {
    "Los Angeles": { "lat": 34.052235, "long": -118.24368 },
    "Atlanta": { "lat": 33.753746, "long": -84.386330 },
    "Miami": { "lat": 25.761681, "long": -80.191788 },
    "New York": { "lat": 40.730610, "long": -73.935242 },
    "Chicago": { "lat": 41.881832, "long": -87.623177 },
    "Seattle": { "lat": 47.608013, "long": -122.335167 },
    "Vancouver": { "lat": 49.246292, "long": -123.116226 },
    "Toronto": { "lat": 43.651070, "long": -79.347015 },
    "Salt Lake City": { "lat": 40.758701, "long":-111.876183 },
    "Denver": { "lat": 39.742043, "long": -104.991531 },
    "Dallas": { "lat": 32.779167, "long": -96.808891},
    "Kansas City": { "lat": 39.099724, "long": -94.578331 },
    "Memphis": { "lat": 35.117500, "long": -89.971107 },
    "Nashville": { "lat": 36.174465, "long": -86.767960 },
    "Columbus": { "lat": 39.983334, "long": -82.983330 },
    "Calgary": { "lat": 51.049999, "long": -114.066666 },
    "Regina": { "lat": 50.445210, "long": -104.618896 },
    "Winnipeg": { "lat": 49.895077, "long": -97.138451 }
  }

    cost_matrix = generate_costs(nodes_coordinates, nodes)
    return nodes, nodes_coordinates, cost_matrix
```

Lista de cidades e coordenadas

```python
def generate_costs(nodes_coordinates, nodes):
  node_cost = []

  for x in nodes_coordinates:
    result = []
    node1 = nodes_coordinates[x]

    for y in nodes_coordinates:
      node2 = nodes_coordinates[y]

      distance = np.sqrt( (node2["lat"] - node1["lat"])  ** 2 + (node2["long"] - node1["long"]) ** 2 )
      distance = distance if distance < 30 else np.inf
      result.append(distance)

    node_cost.append(result)


  cost_matrix = pd.DataFrame(node_cost, index=nodes, columns=nodes)
  return cost_matrix
```

Gerando matriz de custo baseado na distância entre as cidades

Após termos a lista de cidades, coordenadas e a matriz de custo, sorteamos 2 cidades aleatoriamente, uma como origem e outra como destino.

```python
def generate_route(nodes):
  origin = random.sample(nodes, 1)[0]
  destination = random.sample(nodes, 1)[0]

  while (origin == destination):
    destination = random.sample(nodes, 1)

  return origin, destination
```

Sorteando aleatoriamente 2 cidades como origem e destino

Agora todos os dados necessários para começar a resolver o problema, a partir deste momento que começamos a implementar o algoritmo genético.

O primeiro passo é gerar a população inicial, ou seja, criar as primeiras soluções de forma aleatória, para que possamos evoluir estas soluções a cada geração, encontrando a melhor solução.

```python
def initial_population(nodes, origin, destination):
  population = []

  for _ in range(20):

    filtered_nodes = [node for node in nodes if node != origin and node != destination]

    individual = {}
    intermediates = 6
    path = random.sample(filtered_nodes, intermediates)
    path.insert(0, origin)
    path.append(destination)
    individual['path'] = path
    population.append(individual)

  return population
```

Gerando as primeiras 20 soluções aleatoriamente, cada solução precisa passar por 6 cidades além da origem e destino

Após gerar a nossa primeira geração, iremos avaliar os resultados de cada solução, neste caso, a melhor solução é que tenha o custo mais baixo.

O custo total é a soma dos custos entre as cidades, a solução de que tiver o custo total mais baixo, será escolhida como a melhor solução desta geração.

```python
def evaluate(population, cost):
  lowest_cost = None
  best_solution = None

  for index, individual in enumerate(population):
    total_cost = 0

    for location in range(len(individual['path'])-1):
      node = individual['path'][location]
      next_node = individual['path'][location+1]
      total_cost += cost.loc[node, next_node]

    individual['cost'] = total_cost

    if lowest_cost is None or lowest_cost > total_cost:
      lowest_cost = total_cost
      best_solution = index

  return population, lowest_cost, best_solution
```

Calculando o custo total da solução e retornando a melhor solução e o menor custo

Neste momento temos os resultados da nossa primeira geração que foi gerada automaticamente, a partir deste momento, iremos gerar a próxima geração de soluções para tentar encontrar uma solução que tenha um custo mais baixo, mas utilizaremos uma técnica chamada de Elitismo, onde vamos enviar para a nova geração, a melhor solução da geração atual.

```python
  population, lowest_cost, best_solution = evaluate(population, cost_matrix)

  costs.append(lowest_cost)
  solutions.append(population[best_solution])
  print(f"Generation {current_generation} - Lowest cost: {lowest_cost}")

  new_population = []
  new_population.append(population[best_solution])
```

Após ter avaliado a melhor solução da geração atual, enviamos a melhor solução para a próxima geração.

Após termos os resultados da nossa geração atual, é hora de gerarmos a nova geração de soluções.

Para isso devemos escolher algumas soluções da geração atual para serem os "pais" desta nova geração, e para isso, utilizamos uma técnica chamada de torneio, onde sorteamos algumas soluções da geração atual, e fazemos um confronto entre elas, onde a que tiver menor custo será a vencedora. Desta forma escolhemos duas soluções para serem os pais da nova geração.

```python
def tournament(population):
  parents = []
  tournament_participants = random.sample(population, 10)

  while len(parents) < 2:
    winner = min(tournament_participants, key=lambda x: x['cost'])
    tournament_participants.remove(winner)
    parents.append(winner)

  return parents
```

Realizando o torneio para escolher os pais da nova geração de soluções

Após finalizado o torneio, iremos gerar as novas soluções da nova geração, para isso iremos utilizar uma técnica chamada Crossover, onde as soluções irão herdar pedaços das duas soluções escolhidas como pais, além disso cada nova solução tem uma chance de sofrer uma mutação, onde iremos inverter duas cidades que compõem a solução.

```python
# Realiza o cruzamento
def crossover(parents):
  origin = parents[0]['path'][0]
  destination = parents[0]['path'][-1]

  parent1 = parents[0]
  parent2 = parents[1]

  length_parent1 = len(parent1)
  length_parent2 = len(parent2)

  child1 = {}
  child2 = {}

  if min(length_parent1, length_parent2) > 2:

    if min(length_parent1, length_parent2) == 3:
      index = random.randint(0, min(length_parent1, length_parent2) -2)
      path_child1 = parent1 + parent2[index:]
      path_child2 = parent2 + parent1[index:]

    else:
      index = random.randint(1, min(length_parent1, length_parent2) - 2)
      path_child1 = parent1[:index] + parent2[index:]
      path_child2 = parent2[:index] + parent1[index:]

    path_child1 = fix_path(path_child1, origin, destination)
    path_child2 = fix_path(path_child2, origin, destination)

    child1['path'] = path_child1
    child2['path'] = path_child2

    return [child1, child2]

  else:
    return parents
```

Gerando uma nova solução a partir de 2 soluções originárias

```python
# Realiza a mutação
def mutation(path, nodes, origin, destination):
  mutated_path = path.copy()
  length = len(mutated_path)
  filtered_nodes = [node for node in nodes if node != origin and node != destination]

  if length == 2 or length == 3:
    node_chosen = random.sample(filtered_nodes, 1)
    mutated_path.insert(1, node_chosen[0])

  elif length > 3:
    mutated_path.pop(random.randint(1, length-1))

  mutated_path = fix_path(mutated_path, origin, destination)
  mutated = {}
  mutated['path'] = mutated_path

  return mutated
```

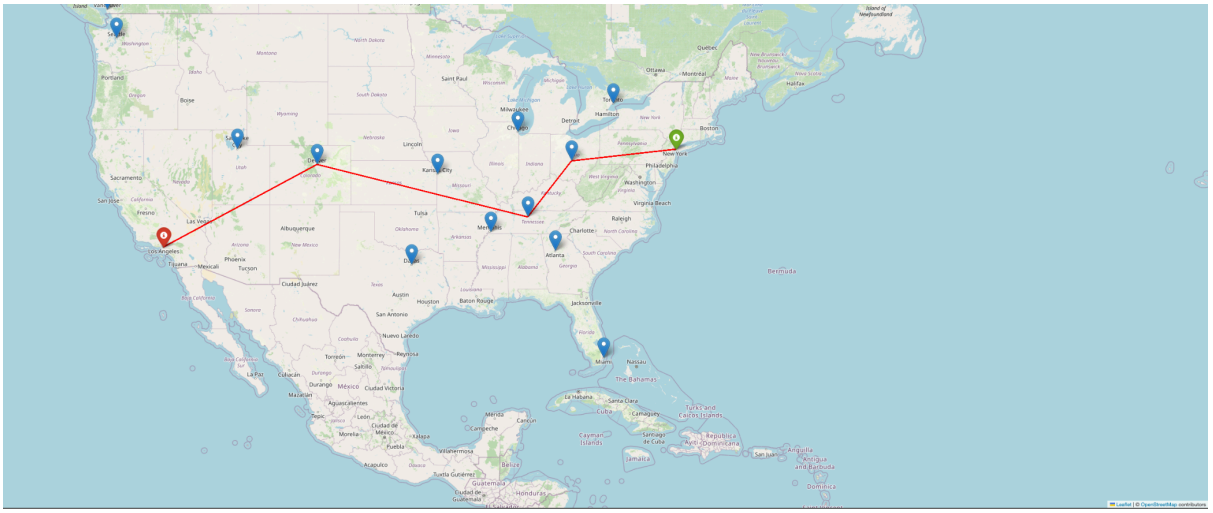
Avaliando se haverá mutação, e se houver, invertendo duas partes da solução

Este passo é executado até termos a quantidade necessária de soluções para substituir a geração anterior.

Após isso realizamos novamente os passos anteriores, para avaliar a nova geração e a partir dela gerar uma nova geração de soluções, até termos um resultado melhor que o anterior.

```python
while current_generation < GENERATIONS:
    population, lowest_cost, best_solution = evaluate(population, cost_matrix)

    costs.append(lowest_cost)

    solutions.append(population[best_solution])

    print(f"Generation {current_generation} - Lowest cost: {lowest_cost}")

    new_population = []
    new_population.append(population[best_solution])

    while len(new_population) < POPULATION_SIZE:
        child = generate_child(population)
        for x in child:
            mutated_child = mutation(x['path'], nodes, origin, destination)
            new_population.append(x)
            new_population.append(mutated_child)

    population = new_population
    current_generation += 1
```

Executando o algoritmo genético por algumas gerações, para otimizar o resultado.



Rota escolhida pelo algoritmo com o custo otimizado

Resultados

Origem	Destino	Custo	Gerações
New York	Los Angeles	44.90	19
Vancouver	Miami	49.93	6
Seattle	Toronto	44.11	3
Calgary	Miami	42.47	2