Problema do Caixeiro Viajante

O clássico problema de caminho mínimo ou caminho mais curto consiste em encontrar o melhor caminho entre dois nós em um grafo, por exemplo. Assim, resolver este problema pode significar determinar o caminho entre dois nós com custo mínimo, ou com o menor tempo de viagem. Em uma rede qualquer, dependendo de suas características, podem existir diversos caminhos entre um par de nós, definidos como origem e destino. Entre os vários caminhos, aquele que possui o menor "peso" é chamado de caminho mínimo. Este peso representa a soma total dos valores dos arcos que compõem o caminho e estes valores podem ser: o tempo de viagem, a distância percorrida ou um custo qualquer do arco.

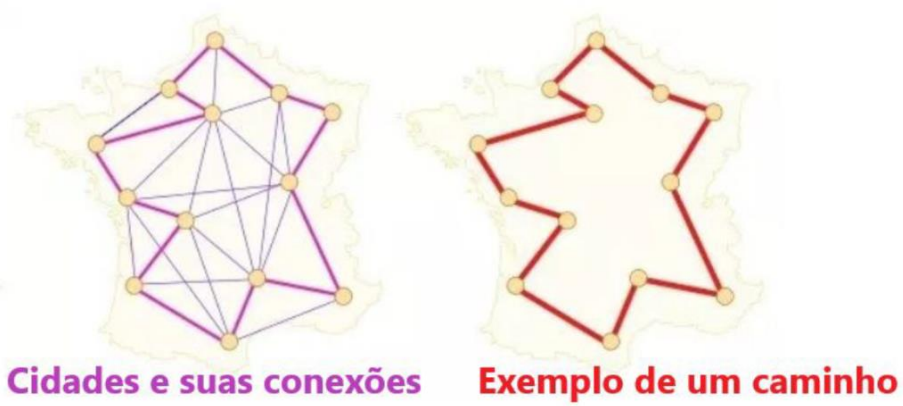
O problema do caixeiro viajante, ramificação do problema de caminho mínimo, consiste em encontrar a menor rota que passe por N cidades conectadas entre elas. É também um clássico problema de otimização combinatória, em que um vendedor deve começar sua jornada em uma cidade e retornar a esta depois de viajar para todas as diferentes cidades da lista, sem importar a ordem entre elas. A grande questão é saber qual é o caminho mais eficiente (lê-se "menor caminho") entre todas as cidades.

O algoritmo de Dijkstra é uma das possíveis soluções para o problema de menor caminho de origem única, e também se encaixa perfeitamente para o problema do caixeiro viajante. Funciona em grafos orientados e não orientados, no entanto, todas as arestas devem ter custos não negativos.

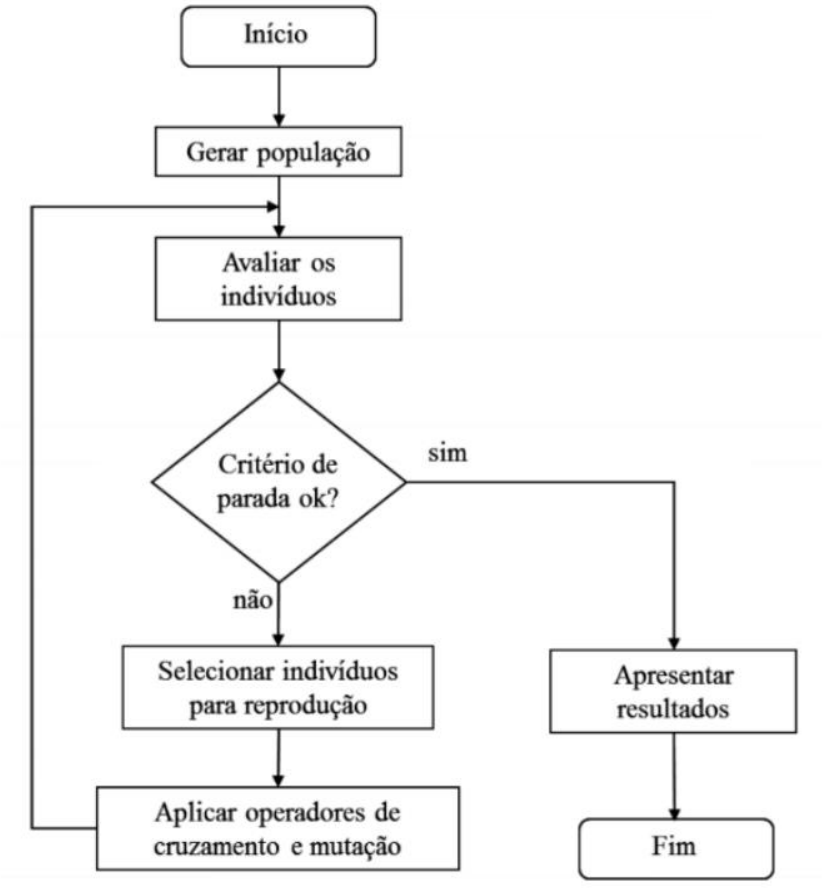
Já o algoritmo de Floyd-Warshall resolve o problema ao determinar o caminho entre todos os pares de nós em um grafo orientado e ponderado. Trata-se de um algoritmo que utiliza matrizes para determinar os caminhos mínimos entre todos os pares de nós da rede. No algoritmo de Floyd são feitas n iterações que correspondem ao número de nós da rede, e a cada iteração corresponde uma matriz D quadrada de ordem n cujos valores são modificados utilizando uma fórmula de recorrência.

Uma das formas de resolver este problema, e a que iremos abordar nesta análise, é usando os conceitos de algoritmo genético, onde inicializamos uma população de indivíduos com genes gerados aleatoriamente e, dado um grafo para representar a distância das cidades e suas conexões, avaliamos a aptidão dos indivíduos por meio de uma função fitness, onde os mais adaptados a resolver o problema serão selecionados para as próximas gerações, passando por

cruzamentos, mutações e seleções para gerarem indivíduos filhos mais adaptados que seus antecessores, segundo a avaliação fitness.



O algoritmo continua a buscar a solução ideal até que um dos critérios de parada sejam satisfeitos, geralmente estes são: a obtenção da solução ótima ou alcance o número máximo de gerações pré-estabelecidas. No nosso caso, não sabemos a solução ótima, então iremos buscar a melhor solução possível até que se alcance o número máximo de gerações do loop.



Exemplo de fluxograma de um algoritmo genético.

O algoritmo em questão foi desenvolvido em Python, no ambiente de desenvolvimento online Google Colab, utilizando o *framework* para computação evolucionária DEAP (Distributed Evolutionary Algorithms in Python). Este *framework* simplifica muito a estruturação e entendimento dos algoritmos genéticos, deixando-os explícitos e suas estruturas de dados, transparentes.

Para aproximar o problema da nossa realidade, este foi modelado em torno de um grafo representativo das cidades ao redor de Araranguá e afins, sendo estas: Criciúma, Tubarão, Lages, Floripa, Sombrio, Laguna, Torres, Caxias do Sul, Porto Alegre, Imbituba, Blumenau, Itajaí, Joinville, Rio do Sul e Caçador. Utilizamos uma matriz de distâncias para registrar a distância em quilômetros de cada uma dessas cidades entre si, onde Araranguá é o ponto de início do grafo e elemento da origem (0;0):

```
matrizDistancia = [ # 0 1  2   3   4  5   6   7  8   9   10  11  12  13  14  15
                    [0,40,77,288,216,26,104,54,255,234,132,347,295,382,310,469],          #Araranguá       0
                    [40,0,65,236,205,67,93,104,266,285,121,333,280,367,268,416],          #Criciuma        1
                    [77,65,0,213,141,103,29,139,329,321,57,269,217,303,245,391],          #Tubarão         2
                    [288,236,213,0,227,285,235,321,217,343,263,223,304,314,129,179],      #Lages           3
                    [216,205,141,227,0,243,123,279,447,462,94,146,100,186,195,410],       #Floripa         4
                    [26,67,103,285,243,0,129,38,229,220,157,369,317,404,333,461],         #Sombrio         5
                    [104,93,29,235,123,129,0,166,357,348,38,249,197,284,268,416],         #Laguna          6
                    [54,104,139,321,279,38,166,0,201,193,193,405,353,440,369,497],        #Torres          7
                    [255,266,329,217,447,229,357,201,0,127,379,440,521,531,346,394],      #Caxias do Sul   8
                    [234,285,321,343,462,220,348,193,127,0,383,568,542,629,474,479],      #Porto Alegre    9
                    [132,121,57,263,94,157,38,193,379,383,0,220,167,254,242,444],         #Imbituba       10
                    [347,333,269,223,146,369,249,405,440,568,220,0,58,103,100,302],       #Blumenau       11
                    [295,280,217,304,100,317,197,353,521,542,167,58,0,93,152,354],        #Itajai         12
                    [382,367,303,314,186,404,284,440,531,629,254,103,93,0,188,326],       #Joinville      13
                    [310,268,245,129,195,333,268,369,346,474,242,100,152,188,0,204],      #Rio do Sul     14
                    [469,416,391,179,410,461,416,497,394,479,444,302,354,326,204,0],      #Caçador        15
                ]
```

Como podemos ver, trata-se basicamente de um *array* de *arrays*, onde o *array* de índice zero trata-se das distâncias de Araranguá até as 16 cidades existentes no modelo (considerando a distância de Araranguá para Araranguá como zero), enquanto que o *array* de índice quinze trata-se das distâncias de Caçador até as 16 cidades listadas até aqui, sendo o primeiro elemento de cada *array* a distância da cidade em questão até Araranguá, o segundo elemento como a distância da cidade em questão até Criciúma, e assim por diante.

Para dar continuidade à explicação da modelagem, é necessário explicar brevemente o funcionamento de alguns componentes da biblioteca DEAP:

```
from deap import creator, base, tools, algorithms
```

Usando o trecho de código acima, importamos a biblioteca DEAP e seus submódulos creator, base, tools e algorithms, onde:

- creator: Permite criar indivíduos e a função de aptidão;
- base: Registra os elementos do algoritmo genético;
- tools: Permite utilizar as funções de operadores genéticos;
- algorithms: Executa o algoritmo genético;

Através do submódulo creator, podemos especificar que tipo de problema queremos resolver, se este é de maximização ou de minimização, de único objetivo ou de múltiplas funções objetivo.

Este submódulo também cria a estrutura dos nossos indivíduos a serem avaliados, definindo seu tipo e sua função fitness, criada anteriormente.

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,)) #Função de Minimização com um único objetivo (obter menor distância)
creator.create("Individual", array.array, typecode='i', fitness=creator.FitnessMin) #cria a estrutura do indivíduo
```

Dito isso, podemos observar acima na primeira linha, a criação da função fitness de Minimização com o objetivo unicamente de obter a menor distância para avaliar os indivíduos. Na segunda linha podemos ver que a estrutura dos indivíduos foi definida, sendo do tipo array.

Enfim, criada a estrutura dos indivíduos, temos que registrar quais os valores que cada gene poderá assumir. Esta função é estabelecida pelos submódulos base e tools.

Primeiramente criamos a base de definições e, com isso, dispomos de uma toolbox, que nada mais é do que uma caixa de ferramentas para este problema, onde serão registrados os objetivos e elementos do algoritmo genético, incluindo a aptidão e a estrutura do indivíduo, criados anteriormente. Em seguida, criamos a estrutura dos genes dos indivíduos, gerados aleatoriamente em um range dado pela variável tamanho=16, mesmo tamanho do array de cada indivíduo, e, consequentemente, mesma quantidade de cidades existentes no problema.

```
toolbox = base.Toolbox() #registra todos os elementos na toolbox
toolbox.register("genes", random.sample, range(tamanho), tamanho)  #os genes são os indices das cidades

toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.genes) # inicializa os individuos com a estrutura criada e os genes(indices) criados acima
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

Estes genes serão os índices das cidades que devemos percorrer, logo, o indivíduo carregará informação da ordem das cidades que deve ser percorrida para se obter uma possível solução satisfatória. A função de aptidão então irá calcular a

distância das cidades representadas pelos índices, na ordem dada pelo indivíduo, partindo sempre da origem (Araranguá, índice 0) e retornando a esta ao final, sendo todas essas viagens calculadas pela função.

Repare no seguinte trecho de código já apresentado:

```
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.genes) # inicializa os individuos
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

Nele, estamos criando e inicializando aleatoriamente alguns indivíduos, inicializando a população com a estrutura de array já definida e com os genes aleatórios que acabamos de definir.

Por fim, registramos e iteramos a população, que será uma lista de indivíduos. Para este problema, definimos uma população grande, de 200 indivíduos.

```
pop = toolbox.population(n=200) #população
```

Quanto à definição dos operadores de crossover, mutação e seleção, a biblioteca abstrai mais ainda toda essa questão e nos apresenta diversas opções incríveis.

Para o crossover, utilizamos o seguinte código:

```
toolbox.register("mate", tools.cxPartialyMatched) #crossover
```

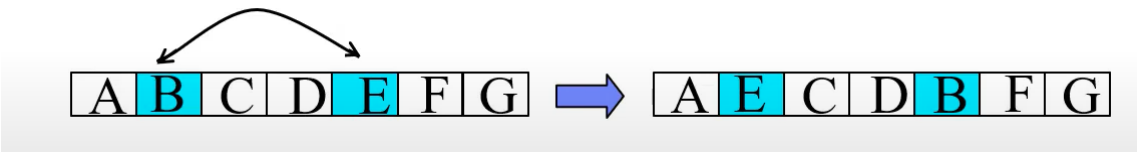
Registramos o crossover com o argumento “mate”, seguido da escolha de algoritmo de crossover que queremos usar. O DEAP possui uma vasta lista de tipos de algoritmos para todos os operadores, mas iremos utilizar o *cxPartialyMatched*, que é próprio para fazer o cruzamento entre soluções que lidam com problemas de permutação, que é o nosso caso.

Esse tipo de crossover realiza, a partir de dois pontos de corte, trocas no sentido de pai 1 para pai 2, e depois no sentido inverso, ou seja, de pai 2 para pai 1, para evitar cromossomos inválidos, gerando dois filhos como de costume em cruzamentos de algoritmos genéticos.

Quanto ao operador de mutação, definiremos através do código:

```
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=0.1)
```

Utilizando o argumento “mutate” para registrar que estamos trabalhando com uma mutação, sendo esta do tipo Order-Based Mutation (mutShuffleIndexes), que basicamente troca o elemento da posição i com o elemento na posição j, de forma que não tenhamos problemas com índices/genes inválidos e/ou repetidos, por se tratar de um problema de permutação. No código acima, o argumento *indpb* indica a probabilidade de um índice ser movimentado, sendo que para tal, utilizamos um valor de 10%.



Para o operador de seleção, utilizamos uma seleção por torneio, dada pelo seguinte código:

```
toolbox.register("select", tools.selTournament, tournsize=2)
```

Com ele, seleciona-se um grupo de k indivíduos aleatórios da população, com k < população, na qual não se utiliza a aptidão dos indivíduos. Após esta seleção, seleciona aquele que possuir a melhor aptidão. Como último parâmetro, utilizamos tournsize = 2 para criarmos torneios 2 a 2.

Quanto à função aptidão, responsável por quantificar quão boa é uma determinada solução, avaliando todos os indivíduos gerados pelo algoritmo, temos o seguinte código:

```
def evalPCV(individual):
    distancia = matrizDistancia[individual[-1]][individual[0]] #Calcula a volta para origem
    for gene1, gene2 in zip(individual[0:-1],individual[1:]):
        distancia += matrizDistancia[gene1][gene2]             #Percorre toda a lista para calcular distancia do trajeto
    return distancia,
```

Nele, definimos a função aptidão dos indivíduos dada pela questão primordial do problema do caixeiro viajante: o menor caminho percorrendo todas as cidades, partindo da origem e retornando a esta ao final, recebendo um incremento na distância a cada movimento feito entre os índices (cidades, genes) dos indivíduos.

Com isso, também devemos registrar esta função, com o argumento “evaluate”, na nossa base de definições já conhecida:

```
toolbox.register("evaluate", evalPCV)
```

Nossa biblioteca também proporciona um módulo de estatísticas para salvarmos nosso progresso com todos os indivíduos.

```
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("mean", np.mean) #média
stats.register("std", np.std) #desvio padrão
stats.register("min", np.min) #mínimo
stats.register("max", np.max) #máximo
```

Com os seguintes códigos, registramos as estatísticas a serem salvas. Na primeira linha, definimos diretamente que queremos salvar as funções de aptidão de cada indivíduo, para calcularmos as estatísticas depois. Em seguida, registramos as estatísticas de média, desvio padrão, mínimo e máximas distâncias de cada indivíduo, para nos auxiliar a entender a curva de evolução mais tarde.

Além desse módulo de estatísticas, também temos um módulo de Hall da Fama, onde podemos armazenar o melhor ou melhores indivíduos de cada geração, como em um ranking.

```
hof = tools.HallOfFame(1) #salva em uma lista o melhor indivíduo da população
```

No nosso problema, armazenaremos apenas o melhor dentre todas as gerações.

A execução em si do nosso algoritmo genético é dada pelo seguinte código:

```
result,log = algorithms.eaSimple(pop, toolbox, cxpb=0.7, mutpb=0.1, ngen=500, stats=stats, halloffame=hof, verbose=True)
```

Onde:

- *pop* é a População;
- *toolbox* é nossa caixa de ferramentas com todas as definições estabelecidas até agora;
- *cxpb* é a probabilidade de crossover, estabelecida como 70%;
- *mutpb* é a probabilidade de mutação, estabelecida como 10%;
- *ngen* é o número de gerações que analisaremos;
- *stats* são nossas estatísticas;
- *hof* é nosso Hall da Fama;
- e o argumento *verbose* como *true* serve para imprimirmos os resultados ao longo da execução.

Esses parâmetros foram estabelecidos através de avaliações empíricas não só com este problema, mas também com avaliações de algoritmos genéticos anteriores. Avaliações estas que nos permitiram enxergar uma eficiência maior na busca pela melhor solução quando se usa uma média probabilidade de crossover, com dois pontos de corte preferencialmente, para obtermos uma variabilidade um pouco maior, tomando cuidado para não aumentar demais esta taxa, que tende a produzir resultados estáveis em um *range* muito amplo, mas que explode os resultados quando passa de um valor crítico arbitrário, pois com alta taxa de cruzamento de estruturas, presume-se que estruturas com boas aptidões possam ser retiradas mais rapidamente na combinação, reduzindo a probabilidade de sucesso.

Quanto à mutação, deve-se usar uma pequena porém não nula taxa de mutação, essencial para produzir indivíduos novos dentro da população e não estagnar os resultados, porém taxas médias ou altas desta podem interferir de forma a criar populações extremamente aleatórias, em contrapartida à mutação nula, que estagna a população.

Utilizamos também uma alta população e uma grande quantidade de gerações, de forma a aumentarmos a probabilidade de sucesso e a rapidez com que este é alcançado (ignorando os tempos ínfimos de execução), com mais diversidade de indivíduos aleatórios a serem cruzados, mutados e selecionados até obter a melhor aptidão.

A fim de não poluir o relatório com uma quantidade irrisória de dados a respeito de todos os indivíduos de cada geração, como a média, a distância máxima, mínima e o desvio padrão de cada população, vamos analisar apenas o melhor indivíduo, dado como solução ideal até o momento, e um gráfico de Aptidão x Geração, onde analisaremos as médias, máximas e mínimas aptidões produzidas.

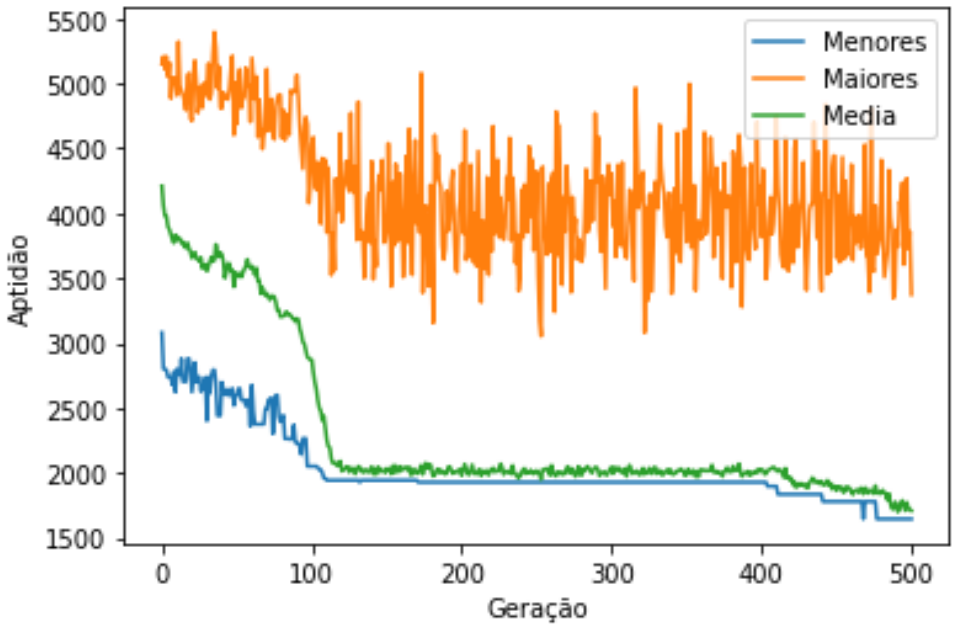
Concluídas estas análises e justificativas, seguimos para os resultados:

```
O melhor indivíduo é:  [Individual('i', [3, 15, 14, 11, 13, 12, 4, 10, 6, 2, 1, 0, 5, 7, 9, 8])]

 Menor distância:  (1646,)
```

Como podemos ver pelo log gerado, o melhor indivíduo obtido teve como melhor distância 1646 km, passando por todas as cidades listadas, dadas pelos índices de 0 a 15, como já comentado.

Quanto ao gráfico, obtivemos o seguinte:



Com ele, é possível concluir ainda que, com os parâmetros utilizados, obtivemos uma estagnação das menores aptidões ao longo de em média 300 gerações, da geração 100 até a 400, onde o *fitness* ficou pouco menor que 2000.

Da geração 400 em diante, as aptidões caíram consideravelmente, nos proporcionando o melhor resultado até agora: 1646 km.

O problema estudado é só um dos vários que podemos resolver utilizando a abordagem única dos algoritmos genéticos, mas este realmente se destacou por mostrar como podemos aplicá-los tanto desafios clássicos quanto em situações cotidianas, como foi o contexto das cidades utilizadas. O potencial para estes algoritmos é impressionante, e descobrir a existência de uma biblioteca como a DEAP foi essencial para vislumbrar a evolução que essa abordagem tem pela frente em nossos cotidianos e nas mais diversas áreas do conhecimento, pesquisa e desenvolvimento.

Link para o Colab do código utilizado:
https://colab.research.google.com/drive/1ejKKv7QnnFXWWnORFUUg-_5Uk6nvNPWV?usp=sharing