Algoritmo genético
Resolvendo problema do caixeiro viajante

Gabriel da Silva Souza

Matrícula: 212050104
Repositório no GitHub: https://github.com/Gabriel-Souza18/AG_caixeiro_viajante

Problema do caixeiro viajante

Suponhamos que um caixeiro tenha que viajar em N cidades, temos que achar o menor caminho em que ele consiga fazer isso, suponha também que tem como ir de uma cidade a todas as outras cidades, sem necessariamente passar em outras.

Exemplo:

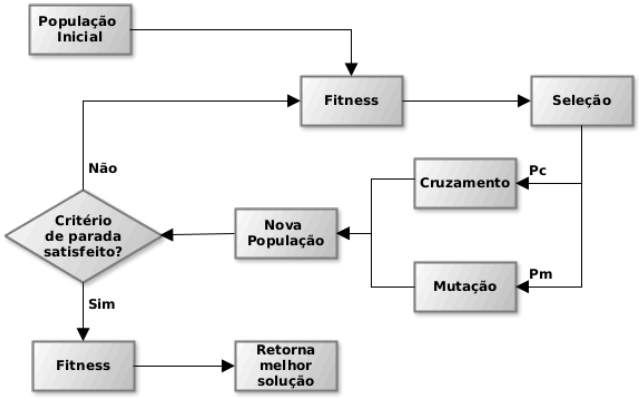
Exemplificando o caso N = 4:

Se tivermos quatro cidades A, B, C e D, uma rota que o caixeiro deve considerar poderia ser: saia de A e daí vá para B, dessa vá para C, e daí vá para D. Quais são as outras possibilidades ? É muito fácil ver que existem seis rotas possíveis:

ABCD
ABDC
ACBD
ACDB
ADBC
ADCB

Algoritmos Genéticos

Algoritmos genéticos são fortemente inspirados na teoria da evolução Darwiniana, baseando-se na sobrevivência do mais apto, e na reprodução desses indivíduos, e em mutações.



O algoritmo possui algumas bases como:

- Cruzamento: Seleciona dois indivíduos de uma geração para misturar seus genes, essa seleção tem que ocorrer de uma forma que faça sentido para o problema, para que o algoritmo chegue numa boa solução, e forme um novo indivíduo, para a próxima geração.
- Mutação: Se apenas tiver cruzamento, provavelmente você chegará em um mínimo ou máximo local, mas a mutação modifica alguns genes de poucos indivíduos, e assim gerando novos caminhos, talvez melhor do que o caminho antigo. importante que essa não tenha muita mutação se não seu algoritmo fica praticamente aleatório.

- Elitismo: Se o melhor indivíduo de uma geração for copiado para a próxima geração, o máximo que ocorre é estagnação, pois nunca piora.

Implementação

Agora que já foi explicado o problema é a maneira que será resolvido, mostrarei como implementei algumas das funções principais do trabalho.

Já na main possui valores que indicam quantos testes serão feitos, quantas gerações cada teste vai possuir e quantos indivíduos cada geração terá.

```
def main():
    num_testes = 5
    num_geracoes = 15
    num_individuos = 20
    chance_mutacao = 5
```

Para gerar a primeira geração, tem uma função usando a biblioteca Random, usando o grafo dado ela retorna um indivíduo.

```
def gerar_individuo_aleatorio(grafo: nx.Graph):
    individuo = {}
    caminho = []
    fitness = 0
    raiz = int(random.uniform(0, len(grafo.nodes())))
    while True:
        for i in range(len(grafo.nodes()) + 1):
            proximo = random.choice(list(grafo.nodes()))
            if proximo not in caminho:
                caminho.append(proximo)
                raiz = proximo
        if len(caminho) == len(grafo.nodes()):
            print("gerou 1 individuo")
            break
    fitness = avaliacao_fitness(caminho, grafo)
    individuo = {'caminho': caminho, 'fitness': fitness}
    return individuo
```

Já as próximas gerações, são feitas a partir de cruzamentos, com os melhores indivíduos tendo mais chances de serem escolhidos, para gerar um novo indivíduo.

Serão gerados num_individuos-1 indivíduos, pois o melhor indivíduo já é copiado para a próxima geração.

```
def geracao_cruzamento(self):
    melhores = []
    melhores = self.selecionar_melhores()
    self.geracao = []
    self.geracao.append(melhores[0])
    self.pesos_cruzamento = pesos_cruzamento(self.npopulacao)
    for i in range(1, self.npopulacao):
```

```
            individuo1 = random.choices(melhores, weights=self.pesos_cruzamento,
k=1)
            individuo2 = random.choices(melhores, weights=self.pesos_cruzamento,
k=1)
            filho = self.cruzamento(individuo1[0], individuo2[0])
            filho["caminho"] = self.mutacao(filho["caminho"])
            self.geracao.append(filho)
        self.geracoes.append(self.geracao)
```

A função de cruzamento recebe dois indivíduos e seleciona pontos de recorte neles, copiando o caminho de um dos pais para a primeira parte, depois colocando o caminho do outro pai, e verificando se não possui duplicatas.

```
    caminho[ponto_corte1:ponto_corte2 + 1] =
individuo1["caminho"][ponto_corte1:ponto_corte2 + 1]

    # Preencher restante do caminho com genes do segundo pai
    posicao_atual = (ponto_corte2 + 1) % len(self.grafo.nodes())
    for gene in individuo2["caminho"]:
        if gene not in caminho:
            caminho[posicao_atual] = gene
            posicao_atual = (posicao_atual + 1) % len(self.grafo.nodes())

    fitness = avaliacao_fitness(caminho, self.grafo)
    filho = {'caminho': caminho, 'fitness': fitness}
    return filho
```

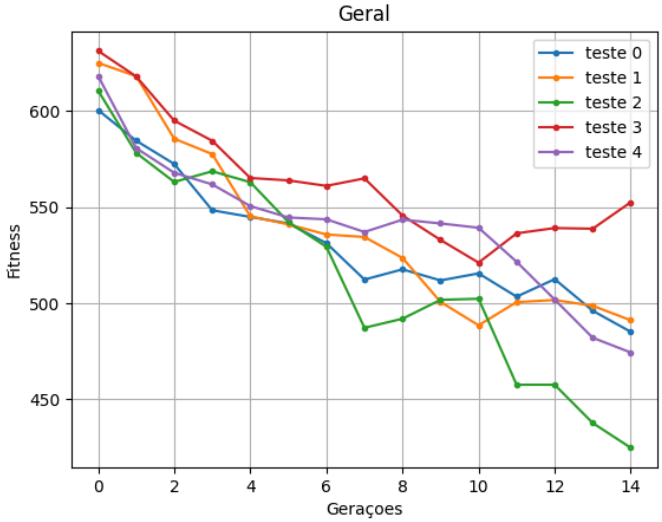
Já na mutação, ela pega um número aleatório e ve se ele é menor que a chance de mutação definida em chanche_mutacao, se for, escolhe dois genes aleatórios e inverte eles

```
    def mutacao(self, caminho: list):
        if random.randrange(start=0, stop=99) < self.chance_mutacao:
            i = random.randrange(start=0, stop=len(self.grafo.nodes())-1)
            while True:
                k = random.randrange(start=0, stop=len(self.grafo.nodes())-1)
                if k != i:
                    break

            auxiliar = caminho[i]
            caminho[i] = caminho[k]
            caminho[k] = auxiliar

        return caminho
```

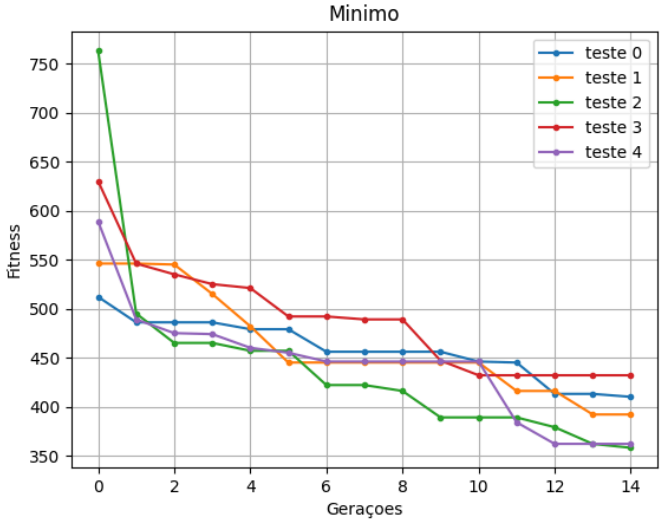
Resultados

Para melhor visualização dos resultados, fiz funções para salvar gráficos, como uma boa maneira de comparar as gerações.

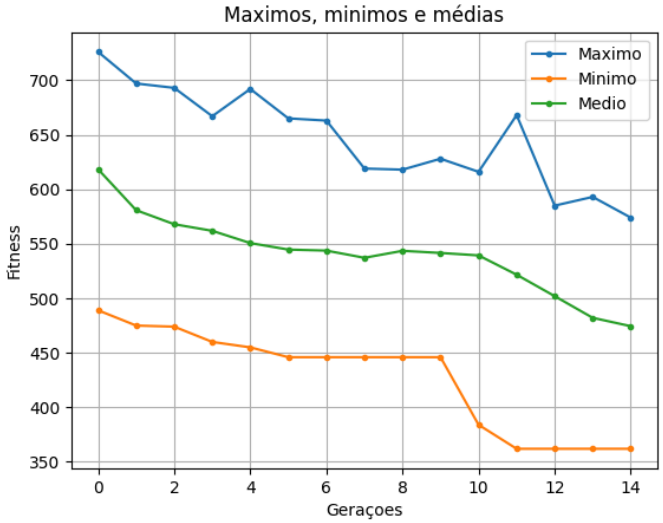
Primeiro fiz um gráfico que imprime a média de cada geração, e cada linha sendo um dos testes realizados, analisando as médias, você pode perceber, que às vezes a média sobe, mas ela sempre baixa com o passar das gerações.



Já o gráfico que mostra o mínimo de cada geração podemos perceber que ele nunca sobe, isso por causa do elitismo, e fica mais estagnado do que a média.



Fiz também um gráfico com um dos testes, mostrando a diferença, do máximo, do mínimo e da média, de cada geração, nele podemos perceber a grande diferença entre o mínimo e o máximo.



Para esses resultados foram usados os valores:

num_testes = 5
num_geracoes = 15
num_individuos = 20
chance_mutacao = 5

E a cada vez que você rodar o código terá resultados diferentes, mas normalmente semelhantes, ainda mais se você aumentar o número de testes.