Aplicação do Algoritmo Genético para o Problema do Caixeiro Viajante

Arthur Tassinari Cabral[1], José Augusto Accorsi[2], Leonardo Broch de Morais[3]

Universidade do Vale do Rio dos Sinos (UNISINOS)
93.020-190 – São Leopoldo – RS – Brasil

arthurtcabral@gmail.com[1], augusto.acorsi@gmail.com[2], leobroch036@gmail.com[3]

Abstract. *This paper talks about the study and the applicability of the Genetic Algorithm for the resolution of the Travelling Salesman Problem. It explores conceptual topics, dealing about the algorithm and the problem, and also talks about this application, that was developed using the Python language.*

Resumo. *Este artigo aborda o estudo e aplicabilidade do algoritmo genético para a resolução do problema do caixeiro viajante. São explorados desde tópicos conceituais, envolvendo o algoritmo e o problema, até a aplicação, desenvolvida utilizando a linguagem Python.*

1. Introdução

Este artigo apresenta o estudo e aplicabilidade do algoritmo genético para a resolução do problema do caixeiro viajante. São explorados, primeiramente, as conceituações referentes ao algoritmo e ao problema, para que, de modo posterior, seja apresentada a utilização do algoritmo.

O objetivo deste artigo é apresentar uma metodologia de resolução de um conhecido problema computacional, buscando agregar tal feito aos conhecimentos obtidos ao longo da disciplina de Inteligência Artificial, na Universidade do Vale do Rio dos Sinos (UNISINOS). Também considera-se como parte do objetivo a exploração dos temas propostos.

Como fontes de consulta, foram utilizados artigos e demais websites do cunho da TI. A própria aplicação também é uma fonte de consulta.

2. Conceituações

2.1 O Problema do Caixeiro Viajante

O problema do caixeiro viajante - em inglês: *The Travelling Salesman Problem* - é um conhecido problema, abordado na área da ciência da computação, que tem como objetivo determinar a menor rota possível para percorrer todas as arestas de um grafo sem repetí-las, bem como utiliza-se tal lógica para lidar com os pontos. Os pontos seriam as representações das cidades, e as arestas das estradas que ligam as cidades de duas-em-duas. Assim sendo, há de se dizer que o problema propõe idealizar uma maneira de localizar o menor caminho possível que um suposto caixeiro viajante utilizaria para a realização do percorrimento em tal cidade.

Em termos envolvendo complexidade de algoritmos, o problema do caixeiro viajante é entendido como sendo um problema da categoria NP-Completo. Os primeiros relatos envolvendo este problema são de 1930, e é tido como um dos problemas mais intensamente estudados no que se refere à otimização algorítmica.

Existem algumas maneiras propostas que visam a resolução ótima deste problema. Uma delas é a de força bruta, que testa um por um dos caminhos possíveis até que se encontre o melhor. Tal abordagem pode ser entendida como inviável quando envolve-se um grafo com um expressivo número amostral de pontos e arestas, pois o número de testes multiplicariam-se, tornando esta metodologia impraticável valendo-se por aspectos envolvendo complexidade de algoritmos. Visto isto, outras formas, envolvendo a aplicação de outros algoritmos, apresentam-se como sendo pertinentes. Uma destas formas é por meio da utilização do algoritmo genético, que tem seu conceito abordado da subseção 2.2 deste artigo.

2.2 Algoritmo Genético

O algoritmo genético - em inglês: *Genetic Algorithm* - é uma técnica de busca bastante estudada na ciência da computação que tem como objetivo encontrar soluções aproximadas em problemas como o do caixeiro viajante, ou seja, problemas que envolvam otimização e busca. É um algoritmo que foi fundamentado por John Holland, que vale-se de técnicas inspiradas na biologia, envolvendo, entre outros conceitos, hereditariedade, mutação e seleção natural.

De maneira a introduzir ao tema, menciona-se alguns componentes genericamente e geralmente presentes no algoritmo genético: A função-objetivo, que equivale ao objeto da otimização, isto é, ao que deseja-se otimizar; O indivíduo, que é uma instância de uma possível solução, que pode ser aprimorada de acordo com a execução do algoritmo; A seleção, que, através da randomicidade, seleciona alguns indivíduos e atribui-lhes probabilidades decrescentes, buscando verificar o mais forte para que este seja mantido; E a reprodução, que busca efetuar a recombinação entre diversos indivíduos para que posteriormente possa-se, por meio da continuação da execução, verificar o mais apto.

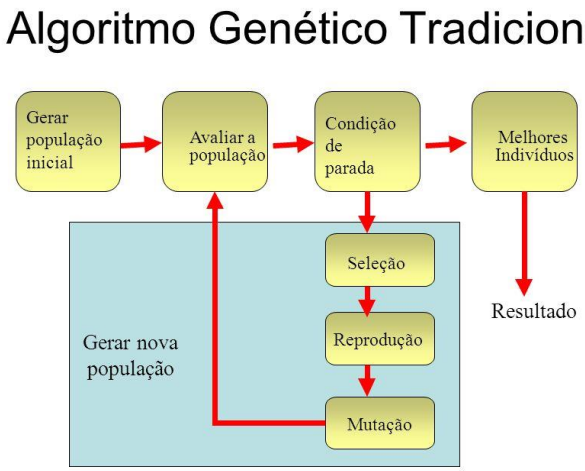

Figura 1. Relacionamentos internos de um algoritmo genético tradicional

3 A Implementação

A implementação realizada para avaliar a aplicabilidade de um algoritmo genético para a resolução do problema do caixeiro viajante foi realizada utilizando a linguagem Python. O algoritmo consiste inicialmente na geração aleatória de um conjunto de possíveis soluções para a instância do problema de forma a calcular a qualidade (Fitness) de cada indivíduo desta

população. Cada indivíduo consiste em um conjunto de arcos que formam o caminho daquela possível solução, e o Fitness do indivíduo é calculado dividindo 1 pela sua distância total.

```python
def createRoute(cityList):
    route = random.sample(cityList, len(cityList))
    return route

def initialPopulation(popSize, cityList):
    population = []

    for i in range(0, popSize):
        population.append(createRoute(cityList))
    return population

```

Figura 2. Funções onde são geradas a população inicial do algoritmo com soluções aleatórias para a instância do problema

O indivíduo do algoritmo está encapsulado dentro da classe Fitness, que possui uma lista de cidades, que se percorrida conforme a ordem da lista, representa o caminho do indivíduo.

```python
class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness= 0.0

    def routeDistance(self):
        if self.distance ==0:
            pathDistance = 0
            for i in range(0, len(self.route)):
                fromCity = self.route[i]
                toCity = None
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
                pathDistance += fromCity.distance(toCity)
            self.distance = pathDistance
        return self.distance

    def routeFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.routeDistance())
        return self.fitness

```

Figura 3. Classe Fitness que representa uma instância de uma possível solução para o problema do caixeiro viajante

Após a etapa de geração da população é realizada então a seleção dos melhores indivíduos, aqueles que possuem o maior valor de Fitness. Está etapa está encapsulada em funções que tem como objetivo ordenar os indivíduos de acordo com seu *fitness* e então selecionar os melhores de acordo com a parametrização de *eliteSize.*

```python
def rankRoutes(population):
    fitnessResults = {}
    for i in range(0,len(population)):
        fitnessResults[i] = Fitness(population[i]).routeFitness()
    return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse = True)

def selection(popRanked, eliteSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index","Fitness"])
    df['cum_sum'] = df.Fitness.cumsum()
    df['cum_perc'] = 100*df.cum_sum/df.Fitness.sum()

    for i in range(0, eliteSize):
        selectionResults.append(popRanked[i][0])
    for i in range(0, len(popRanked) - eliteSize):
        pick = 100*random.random()
        for i in range(0, len(popRanked)):
            if pick <= df.iat[i,3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults

```

Figura 4. Funções que ordenam os melhores indivíduos e escolhem os melhores para serem passados para as seguintes gerações

Após a seleção dos melhores indivíduos da população é realizada a procriação dos melhores indivíduos. A procriação está encapsulado nos seguintes trechos de código.

```python
def matingPool(population, selectionResults):
    matingpool = []
    for i in range(0, len(selectionResults)):
        index = selectionResults[i]
        matingpool.append(population[index])
    return matingpool


def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))

    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):
        childP1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in childP1]

    child = childP1 + childP2
    return child

def breedPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))

    for i in range(0,eliteSize):
        children.append(matingpool[i])

    for i in range(0, length):
        child = breed(pool[i], pool[len(matingpool)-i-1])
        children.append(child)
    return children

```

Figura 5. Funções que realizam a seleção dos indivíduos a serem procriados e também realiza a procriação dos indivíduos

Após a procriação, é então realizada a mutação sobre os indivíduos pertencentes a população. Nesta etapa utilizamos o fator de mutação para avaliar o quanto os indivíduos sofrerão de mutações nesta geração. Para isto para cada indivíduo é percorrido cada arco do caminho e então é gerado um número entre 0 e 1 e este é comparado com o fator de mutação, caso esse número randômico seja menor que o fator de mutação o arco é trocado com outro aleatório.

```python
def mutate(individual, mutationRate):
    for swapped in range(len(individual)):
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))

            city1 = individual[swapped]
            city2 = individual[swapWith]

            individual[swapped] = city2
            individual[swapWith] = city1
    return individual

def mutatePopulation(population, mutationRate):
    mutatedPop = []

    for ind in range(0, len(population)):
        mutatedInd = mutate(population[ind], mutationRate)
        mutatedPop.append(mutatedInd)
    return mutatedPop
```

Figura 6. Funções que realizam a mutação dos indivíduos

```python
def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = breedPopulation(matingpool, eliteSize)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration

def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):
    pop = initialPopulation(popSize, population)
    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))

    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)

    print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))
    bestRouteIndex = rankRoutes(pop)[0][0]
    bestRoute = pop[bestRouteIndex]
    return bestRoute
```

Figura 7. Funções que orquestram a execução do algoritmo

Então para invocar a execução do algoritmo genético devemos invocar o método passando os seguintes parâmetros para execução:

```python
geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01, generations=500)
```

Figura 8. Parâmetros de entrada do algoritmo desenvolvido

Onde *popSize* representa a quantidade de indivíduos gerados a cada geração, *eliteSize* é o parâmetro que indica a quantidade dos melhores indivíduos que serão selecionados a cada geração e irão compor a seguinte, *mutationRate* representa a taxa de mutação e *generations* é o número de gerações que serão criadas ao decorrer da execução do algoritmo.

3.1 Análise dos Resultados

Executando o algoritmo com a instância berlin52 do problema do caixeiro viajante e informando os parâmetros conforme a Figura 8, obtivemos o seguinte resultado:

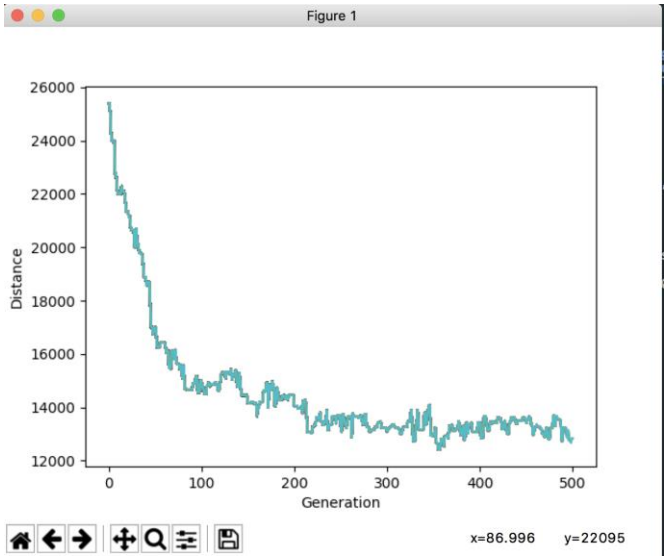

Figura 9. Análise dos resultados obtidos na execução da instância berlin52 com parâmetros informados conforme a Figura 8 deste artigo

Os parâmetros que possuem uma influência direta no desempenho e na qualidade do algoritmo são a taxa de mutação, o número de gerações e o tamanho da elite escolhida a cada geração. Executando o mesmo algoritmo, porém aumentando o número de gerações para 5000, ou seja, iremos gerar 10 vezes mais populações nessa execução, observamos o aumento da qualidade do resultado obtido, porém o tempo de execução aumenta de forma linear.

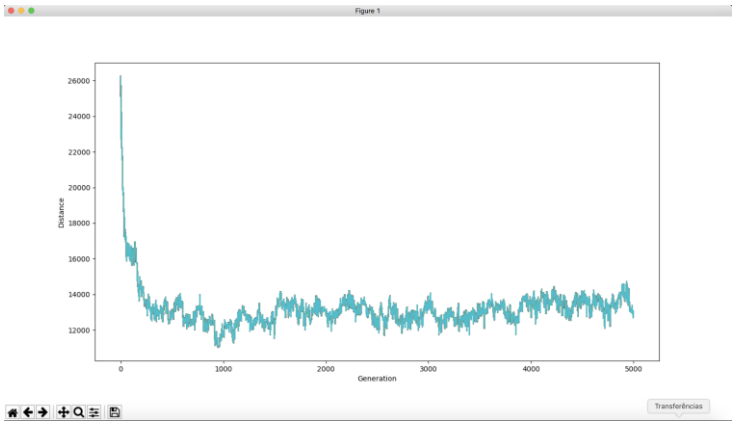

Figura 10. Algoritmo executado em 5 mil gerações

Também pode se observar um resultado bastante distinto executando o algoritmo em 500 gerações porém aumentando o fator de mutação em 10 vezes, ou seja, forçando que os indivíduos sofram mais mutações durante cada geração. Com estes parâmetros a qualidade dos resultados ficou bem abaixo do que o esperado, podendo assim concluir que quando a taxa de mutação possui valores muito altos a solução acaba sendo essencialmente aleatória.

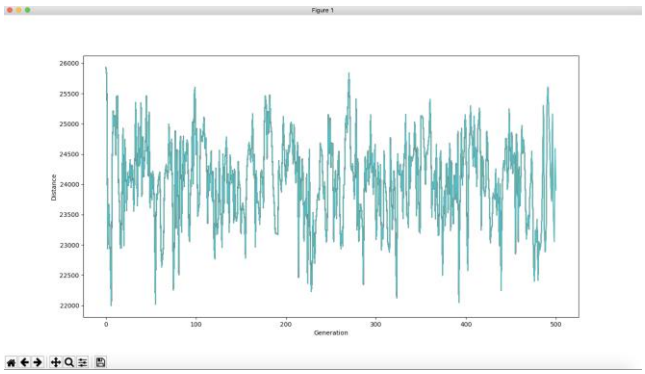

Figura 11. Resultado encontrado com uma taxa de mutação alta

Porém a taxa de mutação ainda precisa estar em um nível adequado para prevenir que uma determinada solução fique estagnado no mesmo valor, diminuindo a taxa de mutação neste algoritmo para 0.001 obtivemos um dos melhores resultados encontrados. Concluindo assim que a taxa de mutação deve ser equalizada em valores que evitem a estagnação dos resultados durante o passar das gerações, mas que não faça com que a geração perca suas qualidades com muitas mutações.

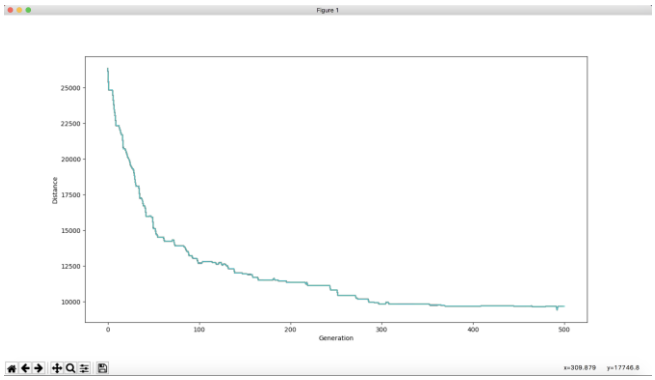

Figura 12. Resultado obtido em 500 gerações com uma taxa de mutação de 0.001

Referências Bibliográficas

Applegate, D. L.; Bixby, R. M.; Chvátal, V.; Cook, W. J. (2006), The Traveling Salesman Problem, ISBN 978-0-691-12993-8. Acesso em 10 abr 2019

Eiben, A. E. et al (1994). "Genetic algorithms with multi-parent recombination". PPSN III: Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: 78–87. ISBN 3-540-58484-6. Acesso em 10 abr 2019

Harik, G. (1997). Learning linkage to efficiently solve problems of bounded difficulty using genetic algorithms (PhD). Dept. Computer Science, University of Michigan, Ann Arbour. Acesso em 10 abr 2019