Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Bacharelado em Sistemas de Informação

ALGORITMO DE FORÇA BRUTA E ALGORITMO GENÉTICO APLICADOS AO PROBLEMA DO CAIXEIRO VIAJANTE

Graziela Maria Felix da Silva

Recife

Julho de 2021

Dedico este trabalho a Deus, a meus pais e a todos que indiretamente me apoiaram nesta jornada.

Resumo

Problemas de otimização são comuns no mundo real, contudo boa parte destes problemas são difíceis de solucionar na teoria, pois se classificam como problemas NP-completos: um exemplo é o clássico “the traveling salesman”, ou Caixeiro Viajante. Nesse sentido, neste trabalho foram abordados dois tipos de algoritmos para a resolução dessa classe de problemas: algoritmo de força bruta e algoritmo genético. Foram utilizados métodos convencionais para a elaboração do algoritmo de força bruta e metaheurística (método heurístico para resolver de forma genérica problemas de otimização, aplicadas a problemas para os quais não se conhece algoritmo eficiente) no algoritmo genético. Os resultados da comparação entre os dois tipos de algoritmos foi satisfatório para as instâncias utilizadas, ressaltando que o uso de metaheurísticas é o meio mais viável nestes casos.

Palavras-chave: caixeiro viajante, força bruta, algoritmo genético, tempo polinomial, tempo exponencial, algoritmos determinísticos, algoritmos não-determinísticos.

1. Introdução

1.1 Apresentação e Motivação

No mundo real é comum encontrar situações onde é necessário a busca da maximização ou a minimização de custos, sejam empacotamento de objetos em containers, a localização de centros distribuidores, o escalonamento e roteamento de veículos, etc. Portanto, é importante a busca por soluções eficientes para cada situação.

O problema de otimização da empresa *FlyFood* tem por objetivo minimizar os custos. Para isso é necessário o cálculo da menor distância que um drone pode percorrer a partir da inserção de coordenadas que mostram o ponto de partida e os pontos de entrega deste equipamento. A leitura das coordenadas deverá ser realizada a partir de uma matriz. A posição de cada ponto na matriz é dada por X_{ij}.

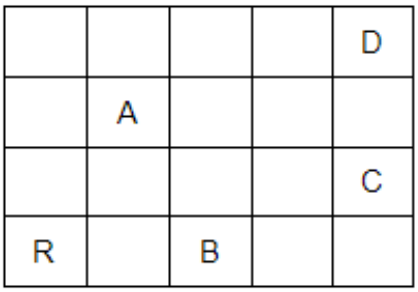

Figura 1. Representação de uma matriz dos pontos
Fonte: PISI2 - Descrição do projeto - Flyfood

Para o cálculo das distâncias entre os pontos será utilizado a *geometria pombalina(ou geometria do táxi)*. Em que, dado dois pontos A e B, com coordenadas (x1,y1) e (x2,y2), respectivamente, calcularemos a distância entre esses pontos fazendo uso da seguinte fórmula: dT = |x2-x1| + |y2-y1|.

Esse tipo de problema é um exemplo de *otimização combinatória*, especificamente uma variável do *problema do caixeiro viajante(PCV)*. O número de rotas *n* possíveis é dada através da equação $\frac{(n-1)!}{2}$, pois se trata de um *PCV simétrico* onde $X_{ij} = X_{ji}$. Isto significa que a distância percorrida entre os pontos é a mesma, não importando a direção. Utilizando a matriz anterior, o número de possibilidades existentes seria: $\frac{(5-1)!}{2}$, cujo resultado é 12 possibilidades.

Ao decorrer deste artigo será possível observar que o aumento do número de pontos de entrega, fará com que a quantidade de $\frac{(n-1)!}{2}$ rotas possíveis aumentem infinitamente. Isso terá relação direta com problemas Np-completo.

Portanto, neste trabalho será desenvolvido uma comparação de resultados entre um algoritmo de força bruta e um algoritmo genético para a resolução desse tipo de problema.

1.2 Objetivos

O objetivo deste trabalho é, utilizando a ferramenta python 3, trazer a implementação de um algoritmo de força bruta e um algoritmo genético que calculam a distância mínima que um drone poderá percorrer passando por todos os pontos de entrega. Em seguida, relacionar os resultados com alguns tópicos da teoria da complexidade computacional, como tempos de algoritmo e classes de complexidade.

1.3 Organização do trabalho

Este trabalho está organizado em: o capítulo 2 é apresentado o referencial teórico, abordando conceitos de algoritmos de tempo polinomial e exponencial, algoritmos determinísticos e não-determinísticos, o problema do caixeiro viajante, algoritmos de força bruta e algoritmos genéticos. O capítulo 3 fornece detalhes dos métodos e ferramentas utilizadas para a elaboração dos algoritmos de força bruta e algoritmo genético. O capítulo 4 apresenta a descrição do passo a passo realizado para criação dos códigos. Em seguida, o capítulo 5 descreve os resultados obtidos e por fim o capítulo 6 apresenta as conclusões.

2. Referencial Teórico

2.1 Problemas NP Completo

Entende-se por problemas “intratáveis” aqueles considerados difíceis ao ponto de não existir um algoritmo **determinístico** capaz de resolvê-los em tempo **polinomial**. Para compreender essa questão e sua relação com problemas NP completo é necessário o entendimento de alguns conceitos:

2.1.1 Algoritmos de tempo polinomial e exponencial

Em computação a importância do tempo de execução de um algoritmo é categórica. Desse modo, uma **Função complexidade de tempo** de um algoritmo expressa o tempo máximo necessário, dentre os possíveis comprimentos de entrada *n* que variam segundo os "tamanhos" dos casos do problema e segundo o "formato" fixado para medir o comprimento de entrada de cada caso.[2]

Definição 1.1 *Algoritmo de tempo polinomial é aquele cuja função complexidade de tempo é O(p(n)) para alguma função polinomial p(n), quando n é o comprimento da entrada.[2] Exemplos: pesquisa binária (O(log n)), pesquisa seqüencial (O(n)), ordenação por inserção ($O(n^2)$), e multiplicação de matrizes ($O(n^3)$).[1]*

Definição 1.2 *Algoritmo de tempo exponencial é aquele cuja função complexidade de tempo não pode ser limitada por um polinômio.[2] Sua função de complexidade é* $O(c^n)$, $c > 1$. *[1] Exemplo:* ***Problema do Caixeiro Viajante (PCV)*** *(O(n!)).[1]*

Com o crescimento do comprimento da entrada *n* de um problema a distinção entre o tempo de execução computacional utilizando algoritmos polinomiais e exponenciais se torna mais evidente. **Problemas NP completo demandam algoritmos de tempo exponencial.** A figura 1 ilustra essa discrepância:

Função complexidade / Tempo	Comprimento n					
	10	20	30	40	50	60
n	0,00001 seg	0,00002 seg	0,00003 seg	0,00004 seg	0,00005 seg	0,00006 seg
n^2	0,0001 seg	0,0004 seg	0,0009 seg	0,0016 seg	0,0025 seg	0,0036 seg
n^3	0,001 seg	0,008 seg	0,027 seg	0,064 seg	0,125 seg	0,216 seg
n^5	1 seg	3,2 seg	24,3 seg	1,7 min	5,2 min	13,0 min
2^n	0,001 seg	1,0 seg	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	0,059 seg	58 min	6,5 anos	3855 séc.	2×10^8 séc.	$1,3 \times 10^{13}$ séc.

Figura 2. Comparação entre os tempos de complexidade-funções polinomial X funções exponenciais.
Fonte: A teoria da complexidade computacional, 1987.

2.1.2 Algoritmos determinísticos e não-determinísticos

Definição 1.3 *Algoritmos determinísticos: pode-se predizer o seu comportamento. Para as mesmas entradas de um problema, teremos sempre as mesmas saídas. Formalmente, um algoritmo determinístico computa uma função matemática; uma função tem um valor único para cada entrada dada, e o algoritmo é um processo que produz este valor em particular como saída.[5]*

Definição 1.4 *Algoritmo não determinístico, é um algoritmo em que, dada uma certa entrada, pode apresentar comportamentos diferentes em diferentes execuções, ao contrário de um algoritmo determinístico. Em outras palavras, em complexidade computacional, algoritmos não determinísticos são aqueles que, a cada passo possível, podem permitir múltiplas continuações. [4]*

Anteriormente, foi introduzido o conceito de problemas “intratáveis”, como algoritmos incapazes de serem resolvidos em tempo polinomial utilizando máquinas determinísticas. No entanto, inúmeros "problemas intratáveis", encontrados na prática, são "decidíveis". Isto significa que é possível resolvê-los, em tempo polinomial, com o

auxílio de um computador "não-determinístico". Estes problemas constituem-se na **classe dos problemas NP.** [2]

2.1.3 Classes de complexidade

Definição 1.5 *Classe P: conjunto de todos os problemas que podem ser* ***resolvidos*** *por algoritmos determinísticos em tempo polinomial.*

Definição 1.6 *Classe NP (non deterministic polynomial): consiste nos problemas que podem ser* ***verificados*** *em tempo polinomial.*

Definição 1.7 *Classe NP completo: consiste nos problemas que demandam tempo exponencial para serem solucionados. Se um deles puder ser resolvido em tempo polinomial então todo problema NP Completo terá uma solução em tempo polinomial. O problema do caixeiro viajante é um exemplo clássico desse tipo de classe.*

2.2 O problema do caixeiro viajante

O problema do caixeiro viajante(ou traveling salesman problem, TSP) pode ser associada a um caixeiro viajante que pretende passar por um conjunto de cidades uma única vez e por fim voltar à cidade inicial, percorrendo a menor distância possível.[6] É um dos problemas mais estudados de otimização combinatória e tem uma grande variedade de aplicações. E foi formulado matematicamente pela primeira vez em 1932, por Karl Menger[7].

O problema diz-se simétrico (STSP) se c(a, b) = c(b, a), para todo a e b, caso contrário, diz-se assimétrico (ATSP). Caso os custos/distâncias entre as cidades cumpram a norma euclidiana, o TSP diz-se euclidiano.[6] No contexto do problema presente neste artigo as distâncias serão dadas por métodos não-euclidianos, especificamente a geometria pombalina (fórmula: dT = |x2-x1| + |y2-y1|). Por ser calculada em módulo, este problema pode ser considerado um TSP simétrico.

2.2.1 Algoritmos de Força Bruta e o problema do caixeiro viajante

Apesar de apresentar um enunciado simples, na prática o problema do caixeiro viajante é um NP-completo. O uso de um método da **pesquisa exaustiva ou de força bruta**, neste caso, consiste no cálculo de todas as possíveis soluções (percursos) e a posterior escolha da solução com melhor custo total.

A partir de um número reduzido de cidades, o algoritmo se torna pouco eficiente, pois para um TSP numa rede completa com n cidades, existem (n−1)! (ou $\frac{(n-1)!}{2}$ no caso simétrico) possíveis soluções para o problema. Na seção resultados será apresentada uma tabela com informações específicas.

2.2.2 Algoritmos genéticos e o problema do caixeiro viajante

Os **algoritmos genéticos (GAs: Genetic Algorithms)** são inspirados na teoria da evolução das espécies de Darwin e na genética. Se utilizam principalmente da probabilidade de um mecanismo de busca *paralela,* cuja característica é realizar diferentes cálculos no mesmo ou em diferentes conjuntos de dados e *adaptativa,* baseado no princípio de sobrevivência dos mais aptos e na reprodução. [8]

Estes princípios são imitados na construção de algoritmos computacionais que buscam uma melhor solução para um determinado problema. No caso do *problema do caixeiro viajante,* a partir de um certo número de cidades, a melhor solução se torna extremamente difícil de encontrar utilizando um algoritmo de *força bruta*, portanto, algoritmos adaptativos como os *GAs* é a opção mais adequada.

Os algoritmos genéticos possuem processos (representação, decodificação, avaliação, seleção, cruzamento, mutação, entre outros) e parâmetros fundamentais (tamanho da população, taxa de reprodução, taxa de mutação e critério de parada). O primeiro conceito é o de representação dos cromossomos, que são as possíveis soluções do problema e podem ser dos tipos: binária, números reais, permutação de símbolos e etc. Neste trabalho foi utilizado a ***permutação de símbolos***. No exemplo:

$$Indivíduo \rightarrow B\ D\ C\ A$$

A ***decodificação*** consiste na construção de uma solução para que o programa o avalie. Neste projeto, é utilizada uma matriz de pontos distintos e suas respectivas posições dentro da matriz. A partir dos pontos são criadas as permutações que serão utilizadas no processo posterior.

A ***avaliação*** se refere ao cálculo do *fitness (aptidão)* de cada indivíduo. No caso do caixeiro viajante, se trata da soma das distâncias dos caminhos de cada indivíduo. É a distância entre cada letra, calculada pela o formula matemática: $dT = |x2-x1| + |y2-y1|$. No exemplo:

$$Indivíduo \rightarrow B\ D\ C\ A \rightarrow 14$$

O processo de ***seleção*** em algoritmos genéticos seleciona indivíduos para a reprodução. E se baseia na escolha dos indivíduos que possuem os melhores *fitness* ou os indivíduos mais aptos, neste caso o indivíduo mais apto é o que possui o menor caminho. Para a realização da seleção existem diferentes métodos, contudo, o utilizado neste projeto foi o ***método da roleta:*** onde cada indivíduo é representado por uma fatia proporcional a sua aptidão relativa. O operador de seleção usado na roleta foi a soma do fitness total da população dividida por cada indivíduo, mais um piso e o valor 0,01. O piso é incremental, seu valor inicial é zero e a cada indivíduo, a partir do segundo, é somado o valor do indivíduo anterior mais 0,01:

$$piso \ + \ \frac{\sum_{j=1}^{n} \int i \rightarrow Soma\ total}{\int i \ \rightarrow individuo} \ + \ 0,01$$

A ideia principal da roleta é a seleção aleatória de pares de cromossomos, que servirão de genitores de dois novos indivíduos criados a partir de seus genes no cruzamento. Quando um indivíduo é selecionado ele é retirado na roleta.

O operador de ***cruzamento (crossover)*** é considerado a característica fundamental dos GAs e ocorre logo após a seleção dos melhores indivíduos. Pares de genitores são escolhidos aleatoriamente da população, baseado na aptidão, e novos indivíduos são criados a partir da troca do material genético. Os descendentes serão diferentes de seus pais, mas com características genéticas de ambos os genitores [8]. Por exemplo:

Pai 1 → *B D C | A E F*
Pai 2 → ***A D B*** ***\| E F C***

Filho 1 → *B D C* ***E F C***
Filho 2 → ***A D B*** *A E F*

A maneira mais simples é escolher apenas um *ponto de corte* aleatório (one-point crossover) e dividir os progenitores em duas partes. A geração dos filhos acontece juntando as partes divididas dos progenitores.

Os cromossomos criados a partir do operador de crossover são então submetidos a operação de ***mutação*** (essa podendo ou não ocorrer dependendo da taxa de probabilidade de mutação). Mutação é um operador exploratório que tem por objetivo aumentar a diversidade na população [8]. O operador de mutação neste projeto aconteceu selecionando dois pontos aleatórios do indivíduo filho e trocando suas posições de lugar.

Por fim, no algoritmo genético os parâmetros controlam o processo evolucionário:

- **Tamanho da População:** é o número de indivíduos que irão compor uma população.
- **Taxa de Crossover:** é a probabilidade de um indivíduo ser recombinado com outro. A taxa utilizada neste projeto foi de 60%, ou seja, apenas sessenta porcento dos indivíduos da população poderão ser selecionados. Com uma taxa muito alta pode ocorrer perda de estruturas de alta aptidão, mas com um valor baixo, o algoritmo pode tornar-se muito lento. [9]
- **Taxa de Mutação:** probabilidade do conteúdo de uma posição/gene do cromossoma ser alterado. De modo geral, as taxas de mutação são muito baixas (<1%), pois taxas altas tornam as alterações essencialmente aleatórias [9]. A taxa utilizada no projeto foi de 0,5%.
- **Número de Gerações:** total de ciclos de evolução de um GAs, ou o número de vezes que o algoritmo irá realizar todos os processos descritos anteriormente até retornar a melhor solução encontrada.

É por meio da evolução de gerações que encontrar uma melhor solução aproximada do problema do caixeiro viajante se torna viável, mesmo que o número de cidades presentes no problema seja extenso.

3. Método

3.1 Algoritmo de Força Bruta

Na primeira parte deste trabalho, foram utilizados para a implementação do algoritmo de força bruta a linguagem de programação Python 3.8.6 e a biblioteca intertools — e seu pacote *permutations* — para a realização das permutações necessárias ao algoritmo.

Além disso, no contexto geral da construção do código, foram empregados conceitos em estruturas condicionais e de repetição, matrizes, dicionários, funções. E realizados testes a partir do uso de arquivos "*.txt"* próprios.

Com o objetivo principal de comparação com o algoritmo genético, já que um algoritmo de força bruta é ineficiente para a resolução do problema central deste trabalho, o caixeiro viajante.

3.2 Algoritmo Genético

A elaboração deste algoritmo tem como objetivo desenvolver um método mais eficiente ao problema do caixeiro viajante, utilizando uma solução adaptativa.

Para a implementação do algoritmo genético foram utilizadas a linguagem de programação Python 3.8.6 e a biblioteca random para a geração de números aleatórios necessários ao algoritmo.

4. Procedimento

Essa seção apresenta o passo a passo realizado para a construção dos algoritmos de força bruta e genéticos para a resolução do problema do caixeiro viajante.

4.1 - Algoritmo de Força bruta

Para a parte inicial do algoritmo de força bruta foi utilizada uma função para realizar a leitura de uma matriz a partir de um arquivo ".txt".

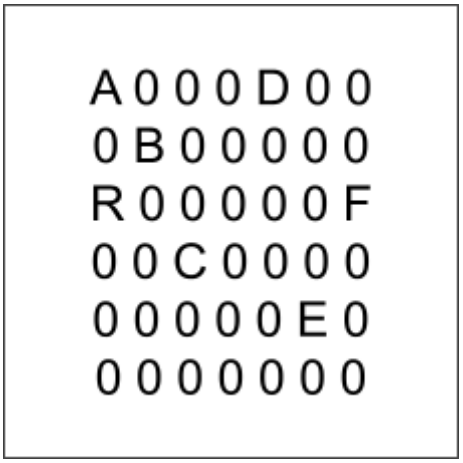

Figura 3. Exemplo de entrada do algoritmo.

Em seguida, foram utilizados laços de repetições para reescrever as informações do arquivo para uma matriz dentro do programa. Então essa matriz foi percorrida item por item, fazendo a verificação dos pontos. Se um item da matriz fosse igual a alguma letra, o ponto X_{ij} e suas coordenadas i e j, dentro de uma tupla, eram armazenados em um dicionário, por exemplo: dicionário = {A = (i, j)}. Os pontos, com exceção do R, também eram armazenados em uma variável tipo lista que posteriormente seria utilizada para a realização das permutações. Quando a verificação da matriz era encerrada, a lista de pontos mencionada anteriormente era ordenada em ordem alfabética e a função *permutations* da biblioteca *intertools* do python iniciava os cálculos das permutações utilizando a lista de pontos mencionada anteriormente. As permutações são armazenadas em outra lista, que chamarei *list_p*.

```
from itertools import permutations
# FORMULA RECURSIVA DA GEOMETRIA DO TAXI

def geoTaxi(x1, y1, x2, y2):
    D = abs((x2-x1))+abs((y2-y1))
    return D

# LEITURA DE MATRIZ A PARTIR DE UM ARQUIVO .TXT

arquivo = open("Casos testes/caso2.txt", "r")
matriz = []
linha = arquivo.readline()
while linha != "":
    elementos = linha.split()
    matriz.append(elementos)
    linha = arquivo.readline()
arquivo.close()
```

```
# BUSCA NA MATRIZ e ARMAZENAGEM DE PONTOS EM UM
DICIONÁRIO

pontos = []
dic_pontos = {}
for i in range(len(matriz)):
    for j in range(len(matriz[0])):
        letra = matriz[i][j]
        if matriz[i][j] == 'R':
            dic_pontos[letra] = (i, j)
        elif matriz[i][j] != '0': #1
            dic_pontos[letra] = (i, j)
            pontos.append(letra)
pontos = sorted(pontos)

# Calculando as permutações
permutacoes = permutations(pontos)
list_p = []
for i in list(permutacoes):
    list_p.append(i)
```

Figura 4. Algoritmo de força bruta - Leitura da matriz e permutações.

Concluído os cálculos das permutações. Os cálculos das distâncias são efetuadas utilizando dois laços for: o primeiro terá o comprimento de *list_p* e indicará a primeira letra das permutações. É chamada a função recursiva *geoTaxi (geometria do táxi)* que calcula a distância entre R e a primeira letra da permutação; o valor é armazenado na variável *soma_dist.* O segundo laço é usado para percorrer cada permutação a partir da primeira letra. Dentro do laço existe uma estrutura de condição *If...else,* onde se o valor de j for igual a posição do último elemento permutação é calculado a distância entre o último ponto e R. As variáveis *ptA* e *ptB* são, respectivamente, o ponto definido por *i e j*, e o ponto posterior *i e j+1*. É feita a chamada da função *geoTaxi* que calcula as distâncias dos pontos. Em seguida, existe mais uma estrutura de condição *If... else* mais um *if,* utilizada para comparar as distâncias totais das permutações e verificar qual a menor. O primeiro if é usado para o caso de a primeira permutação ser a melhor, do contrário, se *soma_dist (*soma das distâncias*)* da permutação calculada for menor que *m_caminho (*melhor caminho*)* armazenado anteriormente, é feita a substituição dos valores e a permutação utilizada no cálculo é armazenado em *best_way* (melhor caminho). Por fim, é imprimido com um *print* o *best_way.*

```
# calculando distancias e verificando o menor caminho

m_caminho = 0
best_way = 'Best way'
for i in range(len(list_p)):
   pt = str(list_p[i][0]) #1
   soma_dist = 0 #1
   soma_dist = soma_dist + geoTaxi(dic_pontos['R'][0],
dic_pontos['R'][1], dic_pontos[pt][0], dic_pontos[pt][1])
   for j in range(len(list_p[i])):
       if j == len(list_p[i])-1:
           pt = str(list_p[i][j])
           soma_dist = soma_dist + geoTaxi(dic_pontos[pt][0],
dic_pontos[pt][1], dic_pontos['R'][0], dic_pontos['R'][1])
       else:
           ptA = str(list_p[i][j])
           ptB = str(list_p[i][j+1])
           soma_dist = soma_dist + geoTaxi(dic_pontos[ptA][0],
dic_pontos[ptA][1], dic_pontos[ptB][0], dic_pontos[ptB][1])
   if i == 0:
       m_caminho = soma_dist
       best_way = str(list_p[0])
   else:
       if soma_dist < m_caminho:
           m_caminho = soma_dist
           best_way = str(list_p[i])

print(best_way)
```

Figura 5. Algoritmo de força bruta - cálculo das distâncias dos pontos e verificação do melhor caminho.

```
('A', 'B', 'D', 'F', 'E', 'C')
```

Figura 5. Algoritmo de força bruta - Saída do código.

Foram utilizados cinco arquivos testes. Cada um deles contendo, respectivamente, quatro, seis, oito, dez e doze pontos de entrega.

4.1 - Algoritmo genético

Para o algoritmo genético, o do tipo de matriz e o processo utilizado para sua leitura é o mesmo que o descrito no algoritmo de força bruta até o ponto de armazenamentos dos pontos e suas coordenadas em um dicionário, além do armazenamento dos pontos em uma lista ordenada em ordem alfabética. Um indivíduo então será representado por uma sequência de letras aleatórias distintas. Exemplo: [A, C, B, D].

Os parâmetros utilizados para a execução do algoritmo foram: tamanho da população = 20, taxa de reprodução = 60%, probabilidade de mutação = 0,5% e critério de parada = 80 iterações. Todos os parâmetros podem ser alterados, contudo o algoritmo apresentou resultados satisfatórios com a taxa de mutação, probabilidade de mutação e o critério de parada igual aos apresentados anteriormente.

```python
# Inicio: 24/06/21 - Fim: 04/07/21
# GRAZIELA MARIA
# RESOLUÇÃO DO PROBLEMA DO CAIXEIRO VIAJANTE UTILIZANDO UM ALGORITMO GENÉTICO

import random

# PARAMETROS DO ALGORITMO GENÉTICO
tamanhoPopulacao = 20
taxaDeReproducao = 60
probabilidadeMutacao = 0.5
criterioParada = 80
```

Figura 6. Algoritmo Genético - Parâmetros.

Na figura 7 está apresentada a função principal do algoritmo genético, onde é criada a população inicial a partir da lista dos pontos ordenados, em seguida é calculado o fitness dessa população inicial. Posteriormente, é iniciada a parte do código que será repetida até atingir o critério de parada(iterações), em que ocorrerão as chamadas das funções.

```python
#ALGORITMO GENETICO
def algoritmoGenetico(pontos, tamanhoPopulacao, taxaDeReproducao, probabilidadeMutacao, criterioParada):
    # CRIANDO POPULAÇÃO INICIAL A PARTIR DO TAMANHO DEFINIDO
    populacao = []
    cont = 0
    while cont < tamanhoPopulacao:
        pt = pontos * 1
        individuo = []
        for i in range(0, len(pontos)):
            id = random.randint(0, len(pt) - 1)
            individuo.append(pt[id])
            pt.remove(pt[id])
        populacao.append(individuo)
        cont += 1

    populacao = sorted(fitness(populacao))
    for i in range(tamanhoPopulacao):
        print("Geração: {} | {}".format("Inicial", populacao[i]))
```

A soma total dos fitness é calculada e a população inicial ordenada do menor fitness para o maior. Em seguida, é calculada a porcentagem de cada indivíduo em relação à soma total, logo após, é calculado seu valor inverso. Então é iniciado o método da roleta, onde os indivíduos são escolhidos aleatoriamente e as chances de escolhas são ditas pela porcentagem inversa do fitness de cada indivíduo. Após retornar os pares dos pais se segue para o crossover, e logo após o ajuste populacional - onde serão eliminados os piores indivíduos aleatoriamente por competição - e finalmente a escolha do melhor indivíduo.

```
geracao = 1
for k in range(criterioParada):
    fitnessTotal = 0 #soma do fitness total de todos os individuos
    for i in range(0, tamanhoPopulacao):
        fitnessTotal = fitnessTotal + populacao[i][0]
    populacao = sorted(populacao)
    #print(fitnessTotal)


    # CALCULANDO PROBABILIDADES PARA O METODO DA ROLETA
    piso = 0
    for i in range(0, tamanhoPopulacao):
        populacao[i].append(round(fitnessTotal/populacao[i][0], 2))
        populacao[i].append(round(piso + populacao[i][2], 2))
        piso = round(piso + populacao[i][2] + 0.01, 2)

    # RETORNANDO OS PARES DE PAIS PARA CROSSOVER
    pais = roleta(populacao, taxaDeReproducao)
    novaPopulacao = crossover(pais, probabilidadeMutacao)
    populacao = sorted(populacao + novaPopulacao)
```

```
    # AJUSTE POPULACIONAL
    populacao = ajustePopulacional(populacao, tamanhoPopulacao)

    # PRINTANDO POPULAÇÕES

    for i in range(tamanhoPopulacao):
        print("Geração: {} | {}".format(geracao, populacao[i]))

    geracao += 1

return populacao
```

Figura 7. Algoritmo Genético - Função algoritmo genético.

Na figura 8, o processo do cálculo dos fitness é o mesmo descrito no algoritmo bruto, com a exceção de que cada fitness de um indivíduo calculado será armazenado em uma lista chamada *fitnessIndividuos* junto com o indivíduo em si. *EX: [14, ['A', 'B', 'C', 'D']]*. E em seguida armazenado em uma lista maior chamada *fitnessPopulacao*. No final do looping será retornada a lista com todos os indivíduos e seus respectivos fitness calculados. Dentro do cálculo do fitness é feita a chamada da função *GeoTaxi()*, descrita na figura 9.

```python
def fitness(populacao):
    fitnessPopulacao = []
    for i in range(len(populacao)):
        fitnessIndividuo = []
        soma_dist = 0
        pt = str(populacao[i][0])
        soma_dist = soma_dist + geoTaxi(dic_pontos['R'][0], dic_pontos['R'][1], dic_pontos[pt][0], dic_pontos[pt][1])
        for j in range(len(populacao[i])):
            if j == len(populacao[i]) - 1:
                pt = str(populacao[i][j])
                soma_dist = soma_dist + geoTaxi(dic_pontos[pt][0], dic_pontos[pt][1], dic_pontos['R'][0],
                                                dic_pontos['R'][1])
            else:
                ptA = str(populacao[i][j])
                ptB = str(populacao[i][j + 1])
                soma_dist = soma_dist + geoTaxi(dic_pontos[ptA][0], dic_pontos[ptA][1], dic_pontos[ptB][0],
                                                dic_pontos[ptB][1])

        fitnessIndividuo.append(soma_dist)
        fitnessIndividuo.append(populacao[i])
        fitnessPopulacao.append(fitnessIndividuo)

    return fitnessPopulacao
```

Figura 8. Algoritmo Genético - Função fitness.

```python
# CALCULO DA GEOMETRIA DO TAXI
def geoTaxi(x1, y1, x2, y2):
    D = abs((x2-x1))+abs((y2-y1))
    return D
```

Figura 9. Algoritmo Genético - Cálculo da geometria do táxi.

A figura 10 representa o método da roleta, que recebe os parâmetros, população e taxa de reprodução. É criada uma cópia da população, pois é necessário a remoção dos indivíduos durante o processo, sem que altere a lista original da população. A taxa é calculada multiplicando a quantidade total de indivíduos na população pela taxa de reprodução dividido por dois e dividido por cem, pois como serão selecionados pares de pais, cem porcento da população é o mesmo que a metade dos indivíduos, ou seja, o para selecionar cem porcento dos indivíduos o *while* rodará 50 vezes. A lógica é a mesma para qualquer porcentagem. O primeiro *for* representa a quantidade de pais que serão selecionados a cada while, a variável *limite* tem o papel de limitar o espaço da roleta pela metade, com o objetivo de selecionar os melhores de indivíduos e o segundo *for* fará o papel de busca pelo indivíduo selecionado na roleta. Os pais selecionados serão armazenados em uma lista chamada *pais* e no fim do *looping* retornados.

```
def roleta(populacao, taxaDeReproducao):
    pop = populacao*1
    cont = 0
    pais = []
    taxa = int(len(populacao) * ((taxaDeReproducao/2) / 100)) #Como são pares de pais 100% é o mesmo que a metade dos in
    #print("A taxa é : ", taxa)
    while cont < taxa:
        pai = []
        for i in range(0, 2):
            limite = round(pop[int(len(pop)/2)][3], 2) #dividi por dois para o limite do random ser o valor do individuo
            roletaPonteiro = round(random.uniform(pop[0][3]-1, limite), 2)
            for j in range(0, len(pop)-1):
                if j == 0:
                    limiteInferior = 0
                    limiteSuperior = pop[j][3]
                    if roletaPonteiro > limiteInferior and roletaPonteiro <= limiteSuperior:
                        pai.append(pop[j])
                        pop.remove(pop[j])
                        break
                else:
                    limiteInferior = pop[(j-1)][3]
                    limiteSuperior = pop[j][3]
                    if roletaPonteiro > limiteInferior and roletaPonteiro <= limiteSuperior:
                        pai.append(pop[j])
                        pop.remove(pop[j])
                        break
        pais.append(pai)
        cont += 1
    return pais
```

Figura 10. Algoritmo Genético - método da roleta.

A figura 11 é representada pelo crossover. A função receberá os parâmetros pais probabilidade de mutação. O parâmetro *pais* será a lista criada no método da roleta e a probabilidade de mutação é a mesma descrita no início do código, igual a 0,5%. A princípio é selecionado aleatoriamente um ponto de corte, este ponto servirá para dividir os pais. O primeiro *filho1* receberá a primeira parte do *pai1* e a segunda parte do *pai2*, já no *filho2* ocorrerá o inverso. Em seguida são chamadas as funções *mutação()* e *organizarFilho()*. Os dois filhos então serão armazenados na lista *novaPopulacao* e por fim são ordenados do menor para o maior, após o cálculo dos seus fitness.

```
# CROSSOVER
def crossover(pais, probabilidaDeMutacao):
    novaPopulacao = []
    pontoCorte = random.randint(1, len(pais[0][0][1])-1)
    for i in range(0, len(pais)-1):
        pai1 = pais[i][0][1]
        pai2 = pais[i][1][1]
        filho1 = pai1[0:pontoCorte]+pai2[pontoCorte:len(pai2)]
        filho2 = pai2[0:pontoCorte]+pai1[pontoCorte:len(pai1)]
        filho1 = mutacao(filho1, probabilidaDeMutacao)
        filho2 = mutacao(filho2, probabilidaDeMutacao)
        orgarnizarFilho(pai1, filho1)
        orgarnizarFilho(pai1, filho2)
        novaPopulacao.append(filho1)
        novaPopulacao.append(filho2)
    novaPopulacao = sorted(fitness(novaPopulacao))
    #print(novaPopulacao)
```

Figura 11. Algoritmo Genético - crossover.

Na figura 12 está representada a função *mutação()*, que receberá os parâmetros filho e taxa de mutação (0,5%). Seu propósito é gerar uma maior aleatoriedade de indivíduos dentro da nova população. É gerado um número aleatório entre 0 e 1. Se a taxa for menor que 0.5 então serão escolhidos dois pontos aleatórios desse indivíduo. Em seguida, é feita a troca da posição de um ponto pela outra. Caso contrário, o filho não será alterado.

```python
# MUTAÇÃO
def mutacao(filho, taxaDeMutacao):
    taxa = random.uniform(0.0, 1.0)
    if taxa < taxaDeMutacao:
        id = random.randint(0, len(filho)-1)
        id2 = random.randint(0, len(filho)-1)
        filho[id], filho[id2] = filho[id2], filho[id]
    return filho
```

Figura 12. Algoritmo Genético - mutação.

A figura 13 apresenta a função *organizarFilho()*, cujos parâmetros são pai e filho. Seu propósito é organizar os filhos que possuem genes repetidos. Primeiro é feita a busca pelas letras repetidas no filho utilizando um *laço for*, em seguida, é encontrada as letras que estão faltando nele. Depois é feita a busca e armazenamento dos índices das letras repetidas do indivíduo filho. Logo em seguida, são realizadas as trocas das letras repetidas pelas que estão faltando.

```python
# REMOVENDO REPETIÇÕES DE LETRAS
def orgarnizarFilho(pai, filho):
    # verificando as letras repetidas e as letras faltando
    letrasRepetidas = [k for k in pai if filho.count(k) > 1]
    letraFaltando = list(set(pai).difference(set(filho)))
    ids = []

    # Verificando os indices das letras repetidas
    if letrasRepetidas == []:
        return filho
    else:
        for i in range(0, len(letrasRepetidas)):
            c = 0
            for x in range(0, len(filho)):
                if filho[x] == letrasRepetidas[i]:
                    c += 1 # contador para não pegar o primeiro item da lista
                    if c > 1:
                        ids.append(x)

    for j in range(0, len(ids)):
        filho[ids[j]] = letraFaltando[j]
    return filho
```

Figura 13. Algoritmo Genético - Removendo genes repetidos.

A função ajuste populacional é iniciada apenas quando houver a junção da nova população com a anterior e feita sua ordenação. Seu objetivo é eliminar os piores indivíduos aleatoriamente, de maneira que a nova população volte a seu tamanho inicial. Os parâmetros da função são: população e tamanho da população. É feita uma competição entre dois indivíduos aleatórios da lista *população*; o pior indivíduo é eliminado.

```python
# AJUSTE POPULACIONAL
def ajustePopulacional(populacao, tamanhoPopulacao):

    while len(populacao) > tamanhoPopulacao:
        tam = len(populacao)
        individuo1 = random.randint(0, tam-1)
        individuo2 = random.randint(0, tam-1)
        if individuo1 != individuo2:
            if populacao[individuo1][0] < populacao[individuo2][0]:
                populacao.remove(populacao[individuo2])
            else:
                populacao.remove(populacao[individuo1])

    return populacao
```

Figura 14. Algoritmo Genético - ajuste populacional.

Por fim, após a função do algoritmo genético atingir o critério de parada, é retornada a melhor solução.

```python
# IMPRIMINDO MELHOR RESULTADO
ag = algoritmoGenetico(pontos, tamanhoPopulacao, taxaDeReproducao, probabilidadeMutacao, criterioParada)
print("Melhor solução encontrada: {} | {}".format(ag[0][0], ag[0][1]))
```

Figura 15. Algoritmo Genético - seleção do melhor indivíduo.

5. Resultados

Para os experimentos realizados neste projeto foram utilizados arquivos testes próprios. Que estão disponíveis para análise juntamente com os códigos dos algoritmos em meu repositório no GitHub [11].

5.1 Algoritmo de Força Bruta

A tabela demonstra os resultados obtidos para cada arquivo de teste. Primeiro, a quantidade de pontos e a quantidade de tempo em segundos necessários para o cálculo do melhor caminho. Em seguida, na tabela é mostrado a quantidade de possibilidades n! que necessariamente o algoritmo de força bruta precisará verificar até encontrar a solução ótima. Perceba que com o aumento da quantidade de pontos o número de possibilidades aumenta exponencialmente, ou mais especificamente, por se tratar do problema do caixeiro viajante, a *complexidade de tempo* é igual a *O(n!)*. Ademais,

segundo os conceitos apresentados na seção referencial teórico, este é um *algoritmo determinístico,* pois as entradas testes apresentam sempre os mesmo resultados, independente de quantas vezes ele seja executado.

Logo após, estão os números de possibilidades para um PCV simétrico, apenas para fins de demonstração. E por fim, as soluções ótimas encontradas para cada caso. Note que para o caso 5, com apenas 12 pontos distintos, não foi possível obter uma solução devido a sua complexidade e baixos recursos de hardware.

Arquivos	Quantidade de pontos	Tempo (segundos)	Possibilidades n!	Possibilidades (n-1)!/2	Solução ótima
caso1.txt	4	0,46	24	3	14
caso2.txt	6	0,6	720	60	22
caso3.txt	8	1	40.320	2.520	38
caso4.txt	10	49	3.628.800	181.440	50
caso5.txt	12	-	479.001.600	19.958.400	-
			Por Graziela Maria		
			Data: 08/05/21		

Tabela 1. Tabela de resultados para o algoritmo de Força Bruta.

5.2 Algoritmo Genético

A tabela 2 demonstra os resultados para o Algoritmo Genético. Nesta tabela foi acrescentado uma nova coluna chamada "tamanho da população", que é um dos parâmetros utilizados neste tipo de algoritmo. Note que para os casos 5 e 6 foram feitas alterações no tamanho da população para alcançar resultados mais satisfatórios. Para cada caso foram realizadas 15 execuções do código. Em cada linha da coluna "Melhor solução" é mostrada a melhor execução de 15 para cada caso. Perceba que os quatro primeiros apresentaram resultados iguais aos resultados do algoritmo de força bruta em algum momento das execuções do código. Vale ressaltar que para o tipo de instância desse problema, que utiliza letras do alfabeto para compor os caminhos, o limite máximo de caminhos é igual ao caso 6. Contudo, pode ser utilizado qualquer outro tipo de símbolo como representação de um ponto (cidade).

Trazendo novamente os conceitos apresentados na seção *referencial teórico,* este é um algoritmo *não-determinístico*, pois apresenta comportamentos diferentes em diferentes execuções, ao contrário de um algoritmo *determinístico.* Além disso, ainda que este problema, em suma, seja classificado como um algoritmo de *tempo exponencial O(n!),* com a utilização do algoritmo genético ele passa a ser tratado como *tempo polinomial;* ou seja é um algoritmo que pode ser resolvido em tempo polinomial,

com o auxílio de um computador "não-determinístico". Estes problemas constituem-se na **classe dos problemas NP.** [2]

Arquivos	Quantidade de pontos	Tamanho da população	Tempo (segundos)	Possibilidades n!	Possibilidades (n-1)!/2	Melhor solução
caso1.txt	4	20	0,60	24	3	14
caso2.txt	6	20	0,80	720	60	22
caso3.txt	8	20	0,90	40.320	2.520	38
caso4.txt	10	20	1	3.628.800	181.440	50
caso5.txt	12	100	1,20	479.001.600	19.958.400	58
caso6.txt	25	150	2,30	1.551121e+25	3.102242008666197e+23	126
				Por Graziela Maria		
				Data: 10/07/21		

Tabela 2. Tabela de resultados para o algoritmo Genético

Dessa maneira, é possível notar já nos primeiros experimentos com o algoritmo genético que ele foi capaz de alcançar resultados satisfatórios.

6. Conclusão

O objetivo deste trabalho era trazer a implementação de um algoritmo de força bruta e um algoritmo genético para a resolução de um problema de rotas de um drone. Esse tipo de problema é classificado como o clássico PCV (problema do caixeiro viajante). E também trazer conceitos fundamentais da computação para o entendimento das diferenças entre os dois tipos de algoritmos e sua eficiência.

Como esperado, os resultados do algoritmo genético foram muito melhores que o algoritmo de força bruta, justamente porque para o tipo de problema tratado neste trabalho um *algoritmo determinístico* como o de força bruta se torna inviável para entradas muito grandes. No entanto, o AG se mostrou eficiente para as entradas testes utilizadas, por seu caráter *não-determinístico*.

Desse modo, conclui-se que problemas da classe NP-completos ainda que considerados “intratáveis” por não existir um *algoritmo determinístico* capaz de resolvê-los em *tempo polinomial.* Podem ser no mínimo "decidíveis" e de *tempo polinomial* utilizando *algoritmos não-determinísticos.* Para trabalhos futuros, existem outros métodos para a

resolução de problemas NP-completos, além dos AGs, enquanto estes podem ainda ser melhorados utilizando diferentes parâmetros, com o intuito de encontrar novas soluções.

Referências Bibliográficas

1. LOUREIRO, Antonio Alfredo Ferreira. Teoria de Complexidade. **UFMG**, 2008. Disponível em: http://www.decom.ufop.br/menotti/paa111/slides/aula-Complexidade-imp.pdf
   Acesso em 23 de dez. de 2020.

2. DE ABREU, Nair Maria Maia. A Teoria da Complexidade Computacional. **R. mil. Cio e Tecno/.**, Rio de Janeiro, 4(1):90-95, jan./mar. 1987. Disponível em: http://rmct.ime.eb.br/arquivos/RMCT_1_tri_1987/teoria_complex_comput.pdf
   Acesso em: 23 de dez. de 2020.

3. ANÁLISE de Complexidade. **UFMG.** Disponível em: https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/complexity.pdf
   Acesso em 23 de dez. de 2020.

4. ALGORITMO não determinístico. **Wikipedia**, 2017. Disponível em: https://pt.wikipedia.org/wiki/Algoritmo_n%C3%A3o_determin%C3%ADstico
   Acesso em 23 de dez. de 2020.

5. ALGORITMO determinístico. **Wikipedia**, 2017. Disponível em: https://pt.wikipedia.org/wiki/Algoritmo_determin%C3%ADstico
   Acesso em 23 de dez. de 2020.

6. OLIVEIRA, André Filipe Maurício de Araújo. Extensões do Problema do Caixeiro Viajante. **UC**, 2015. Disponível em: https://estudogeral.sib.uc.pt/bitstream/10316/31684/1/
   Acesso em 08 de mai. de 2021.

7. ROBERTSON , J J O'Connor. Karl Menger. **MacTutor,** 2014**.** Disponível em: https://mathshistory.st-andrews.ac.uk/Biographies/Menger/
   Acesso em 08 de mai. de 2021.

8. PACHECO, Marco Aurélio Calvalcanti. Algoritmos Genéticos: Princípios E Aplicações. **ICA: Laboratório de Inteligência Computacional Aplicada**, Rio de Janeiro. Disponível em: http://www.inf.ufsc.br/~mauro.roisenberg/ine5377/Cursos-ICA/CE-intro_apost.pdf
   Acesso em: 09 de jul. de 2021.

9. CARVALHO, André Ponce de Leon F. de. Algoritmos Genéticos. **Department of Computer Science.** Disponível em: https://sites.icmc.usp.br/andre/research/genetic/
   Acesso em 09 de jul. de 2021.

10. CUNHA, Hugo Gustavo Valin Oliveira da. Algoritmo Genético e Algoritmo de Vaga-lumes aplicados ao Problema do Caixeiro Viajante. **UNIVERSIDADE FEDERAL DE UBERLÂNDIA.** Uberlândia, Brasil, 2019. Disponível em: file:///C:/Users/enterprise/Desktop/BOOKS/Diversos/AlgoritmoGen%C3%A9ticoAlgoritmo.pdf
    Acesso em: 10/07/21.

11. DA SILVA, Graziela Maria Felix. Repositório do trabalho. **GitHub.** Recife, PE. 2021. Disponível em: https://github.com/Grazifelix/Jorney_Information_Systems_UFRPE/tree/main/Flyfood_2021.1 Acesso em: 10/07/21.