Universidade Tecnológica Federal do Paraná

Tópicos em Análise e Projeto de Algoritmos

Projeto Final TSP de 100 estrelas

Bruno Keller Margariteli **2150883**
Guido Margonar Moreira **2150948**

Professor:

Dr. Luiz Fernando Carvalho

Data de submissão : 26/06/2023

1 Introdução

O Problema do Caixeiro Viajante é um desafio clássico da área de otimização combinatória, que busca encontrar o caminho mais curto possível entre um conjunto de cidades, visitando cada uma delas uma única vez e retornando à cidade de origem. Essa problemática tem aplicação em diversos contextos do mundo real, como roteirização de entregas, planejamento de rotas logísticas e até mesmo em explorações espaciais.

Neste projeto final de Tópicos em Análise e Projeto de Algoritmos, abordaremos o Problema do Caixeiro Viajante em um cenário incomum e fascinante: a viagem entre 100 estrelas, com o objetivo de começar e terminar no nosso próprio Sol. Imagine-se em uma nave espacial, lançando-se ao espaço sideral, com a missão de visitar todas as estrelas previamente selecionadas, retornando ao ponto de partida inicial, percorrendo o menor caminho possível.

O projeto visa explorar essa questão complexa, cujo objetivo é encontrar a rota ótima entre as estrelas, minimizando a distância total percorrida. Para tanto, utilizaremos técnicas avançadas de otimização e algoritmos específicos para solucionar o Problema do Caixeiro Viajante em um espaço de busca tão vasto.

As informações necessárias para realizarmos este projeto foram fornecidas pela universidade de Waterloo. É importante ressaltar que, para um Problema do Caixeiro Viajante com 100 estrelas, temos um total de 100! possíveis soluções.

2 Descrição do Problema

O programa Gaia da Agência Espacial Europeia (ESA) tem como missão criar um mapa tridimensional preciso e abrangente da nossa galáxia. As bases de dados Gaia DR1 e Gaia DR2 já proporcionaram uma vasta quantidade de informações astronômicas, incluindo posições aproximadas de mais de 2 milhões e 1,33 bilhão de estrelas, respectivamente.

Utilizando os dados da ESA, a Universidade de Waterloo, no Canadá, desenvolveu 12 instâncias do problema do caixeiro viajante (Travelling Salesman Problem - TSP), abrangendo de 100 até 1,33 bilhão de estrelas em um ambiente tridimensional. Nesses cenários, a viagem entre as estrelas é medida por distâncias euclidianas em linha reta, arredondadas para o parsec mais próximo.

Os dados foram disponibilizados para a comunidade, incentivando a busca por soluções ótimas. Aqueles que encontrarem as melhores soluções podem compartilhar seus resultados com a Universidade de Waterloo. Esse trabalho destaca a colaboração entre a ESA e instituições acadêmicas na busca pela compreensão e exploração do cosmos.

3 Objetivo

O objetivo deste projeto é investigar e analisar o Problema do Caixeiro Viajante aplicado a um contexto espacial, utilizando um conjunto de dados fornecido pela Agência Espacial Europeia (ESA). O foco principal é explorar o Problema do Caixeiro Viajante com 100 estrelas, onde o objetivo é encontrar a rota mais eficiente, iniciando e terminando no Sol, com base em informações tridimensionais de posicionamento das estrelas.

Nossa meta é estudar diferentes abordagens algorítmicas, técnicas de otimização e métodos de busca para resolver esse desafio complexo. Buscaremos a minimização da distância total percorrida durante a viagem estelar.

Ao final deste relatório, esperamos apresentar uma análise crítica dos resultados obtidos, avaliar a eficácia das estratégias adotadas e discutir possíveis melhorias e limitações encontradas no contexto do problema.

4 Metodologia

Para solucionar este problema escolhemos uma abordagem heurística chamada Algoritmo Guloso, que busca resolver problemas de otimização fazendo escolhas locais ótimas em cada etapa, na esperança de obter uma solução globalmente ótima. Junto com este algoritmo, também aplicamos o algoritmo 2-opt, que procura otimizar uma solução existente, buscando uma rota mais curta e eficiente.

Ao trabalharmos com estes dois algoritmos, podemos encontrar rapidamente encontrar um caminho a percorrer, para assim optimizá-lo até alcançarmos uma eficiência ideal.

4.1 Algoritmo Guloso

O algoritmo guloso é uma abordagem heurística que busca resolver problemas de otimização fazendo escolhas locais ótimas em cada etapa, na esperança de obter uma solução globalmente ótima. Ele recebe esse nome devido à sua natureza "gananciosa", pois faz escolhas imediatamente vantajosas em cada passo, sem considerar as consequências futuras.

O algoritmo guloso começa com uma solução vazia e, a cada iteração, seleciona a melhor opção disponível no momento. Essa escolha é feita com base em uma função de critério, que avalia a qualidade de cada possível escolha de acordo com a estrutura do problema.

Uma característica fundamental do algoritmo guloso é que ele não reconsidera as decisões tomadas anteriormente. Isso significa que ele pode não garantir uma solução globalmente ótima, já que uma escolha localmente ótima pode levar a um resultado sub-ótimo no final.

Apesar de sua simplicidade e eficiência computacional, o algoritmo guloso tem limitações e pode levar a soluções aproximadas, mas não necessariamente ótimas, para certos problemas de otimização. No entanto, em muitos casos, ele fornece soluções satisfatórias e é amplamente utilizado em uma variedade de domínios, como problemas de roteamento, alocação de recursos e seleção de itens.

4.2 Algoritmo 2-opt

O algoritmo 2-opt é uma heurística de melhoria local utilizada para resolver o Problema do Caixeiro Viajante ou outros problemas relacionados ao roteamento. O objetivo do algoritmo é otimizar uma solução existente, buscando uma rota mais curta e eficiente.

O algoritmo 2-opt opera realizando trocas entre arestas em uma rota inicial. Ele começa com uma solução inicial, que pode ser gerada por meio de um algoritmo guloso ou outra heurística. Em seguida, o algoritmo examina todas as possíveis trocas de duas arestas e avalia se a troca resultante reduz a distância total percorrida.

A troca é feita removendo duas arestas da rota atual e substituindo-as por outras duas, de forma que a rota resultante não contenha cruzamentos. Essa operação é chamada de "2-opt swap". O algoritmo repete esse processo para todas as combinações possíveis de trocas de arestas e escolhe a troca que resulta em uma redução máxima na distância percorrida.

O processo de repetir as trocas de arestas é iterado até que não seja possível encontrar mais melhorias na solução atual. Nesse ponto, o algoritmo 2-opt retorna a solução otimizada, que espera-se ser uma solução localmente ótima.

Embora o algoritmo 2-opt não garanta uma solução globalmente ótima para o Problema do Caixeiro Viajante, ele é eficiente e tende a melhorar significativamente a solução inicial, reduzindo a distância total percorrida. É uma técnica comumente utilizada em problemas de roteamento, oferecendo uma abordagem simples e rápida para melhorar soluções aproximadas.

4.3 Mapa 3D

Para visualizarmos melhor o problema e as soluções encontradas, utilizando o Blender (software open source com inúmeras funcionalidades, das quais utilizamos os modelos 3d e a animação por keyframes) criamos um mapa 3D com todas as estrelas em suas respectivas coordenadas e colocamos o caminho pelo qual o algoritmo seguiu. Assim, é possível ter uma ideia visual mais clara do caminho seguido pelo algoritmo.

5 Código Fonte

Abaixo encontra-se o código em Python do algoritmo guloso. Primeiramente, ele importa a biblioteca **numpy** (usada mais à frente). Em seguida, o arquivo **posStars.txt** é aberto; ele contém as coordenadas de cada estrela em suas linhas. Então, o código salva todas as posições em cada linha, agrupadas em grupos de 3 (x, y, z), dentro do vetor **pos**

Com isso feito, é definida a função **calcdist**, que recebe os índices das estrelas de início e fim para, por meio da função **np.linalg.norm** da biblioteca **numpy**, calcular a distância euclidiana entre as estrelas dos índices passados.

A partir da linha 16, é criada a função **getMenor**. Ela recebe como parâmetros o índice da estrela atual, um vetor com as estrelas não exploradas e uma variável inicializada por padrão como lista vazia para os índices a serem ignorados (no caso, ignoramos apenas o índice de início, pois não utilizamos profundidade neste algoritmo). A função calcula a distância entre a estrela na posição **start** em relação a todas as demais no vetor **Astar**, usando a função **calcdist**. Por fim, atribui o índice da menor distância dentro da variável **menor**. Essa função retorna o índice e a distância da estrela mais próxima da atual.

Em seguida, o código coloca essas funções em prática. O vetor **Astar** é criado com os índices de 0 a 99. O vetor **starsF** é criado para armazenar os índices do caminho percorrido, começando na posição zero do sol. O algoritmo guloso começa pegando o primeiro elemento, onde começa a rota, que é colocado na variável **last**, que guarda a última posição visitada.

O algoritmo é executado enquanto a lista **starsA** tiver algum elemento. O índice da menor distância local é obtido e eles são escritos no terminal. **last** é atualizado para esse novo índice. **starsA** remove o objeto já explorado, e **starsF** recebe a nova parte do caminho. No fim desse **while**, só falta voltar para o início,

então é feito o **append** da posição do sol no **starsF**. O caminho é exibido no terminal, a distância total é calculada e exibida usando um **for** passando por todo o **starsF**. Por fim, o caminho é salvo em um arquivo chamado **Guloso.txt**.

```
import numpy as np

#Busca em profundidade

#Abrir arquivo de posi  es
with open("posStars.txt") as f:
  pos = []
  for line in f: # read rest of lines
      pos.append([float(x) for x in line.split()])

#Fun  o calcular dist ncia
def calcdist(start, end):
  return np.linalg.norm([x1-x2 for x1, x2 in zip(pos[start] , pos
   [end])])

#Fun  o calcular dist ncia
def getMenor(start,Astar,ig = []):
  ig.append(start)#n  o tentar voltar para o come o

  menor = Astar[0]
  for i in Astar:
    dm = 0
    dn = 0

    if i not in ig:
      dm += calcdist(start,menor)
      dn += calcdist(start,i)
      if dn < dm:
        menor = i
        dm = dn
  return [menor, dm]


starsA = list(range(0,100))#estrelas Inesploradas (em Aberto)
starsF = [0]# estrelas exploradas

#ALGORITMO GULOSO
#Pega primeira posi  o
last = starsA.pop(0)
#Enquanto existirem estrelas para explorar
while starsA:
  #print("===\n")
  menor = getMenor(last,starsA)
  print("menor opcao local:")
  print(menor)
  #print("[",calcdist(last , menor[0]),"]")
  last = menor[0]
  starsA.remove(menor[0])
  starsF.append(menor[0])
#Volta pro sol
starsF.append(0)
print("Caminho escolhido: ",starsF)

#calcula dist ncia percorrida pelo algoritmo
dist = 0
x = starsF[0]
for i in starsF:
```

```
  dist += calcdist(x, i)
  x = i
print("Dist ncia do algoritmo",dist)

#Salva caminho em Arquivo
file = open('Guloso.txt','w')
for i in starsF:
  file.write(str(i)+" ")
file.close()
```

O Algoritmo 2-opt, a princípio, reutiliza todo o código do algoritmo guloso. No entanto, foram adicionadas 3 funções: **totaldist**, que calcula a distância total ao percorrer a rota fornecida; **optSwap**, que inverte a parte interna de um vetor **ra** dentro do intervalo **i** e **j** fornecido; e, por fim, a função **twOpt**.

A função **twOpt** recebe a rota antiga e tenta melhorar a rota por meio de um **while** que verifica se houve melhoria usando a variável **improved**, que é constantemente atribuída como **False**. Para a melhoria, são utilizados **swaps** que usam dois fors para ir da posição **i+1** até o tamanho da rota - 1, onde i vai de 0a até até o tamanho da rota -1. Após realizar essa inversão, ele verifica se a rota criada teve melhoria na distância. Caso sim, ele atualiza a rota passada, atualiza a melhor distância, define **improved** como **True**, quebra os **for** e repete o processo. Porém, caso ele passe pelos dois **for** fazendo **swaps** com todas as possibilidades e não consiga melhoria, ele sairá do **while** e retornará a nova rota melhorada.

Com isso, basta chamar essa função em **StarsF** depois de realizar o algoritmo guloso, exibir e salvar os dados no arquivo **2opt.txt**.

```
import numpy as np

#Busca em profundidade

#Abrir arquivo de posi  es
with open("posStars.txt") as f:
  pos = []
  for line in f: # read rest of lines
      pos.append([float(x) for x in line.split()])

#Fun  o calcular dist ncia
def calcdist(start, end):
  return np.linalg.norm([x1-x2 for x1, x2 in zip(pos[start] , pos
   [end])])

def totaldist(rota):
  dist = 0
  x = rota[0]
  for i in rota:
    dist += calcdist(x, i)
    x = i
  return dist

#Fun  o calcular dist ncia
def getMenor(start,Astar,ig = []):
  ig.append(start)#n  o tentar voltar para o come o

  menor = Astar[0]
  for i in Astar:
    dm = 0
    dn = 0
```

```
    if i not in ig:
      dm += calcdist(start,menor)
      dn += calcdist(start,i)
      if dn < dm:
        menor = i
        dm = dn
  return [menor, dm]

def optSwap(ra, i, j):
  nr = ra.copy()
  nr[(i+1):j] = nr[(i+1):j][::-1]
  return nr
def twOpt(rotAntiga):
  improved = True
  while improved:
    improved = False
    bestd = totaldist(rotAntiga)
    for i in range(0,len(rotAntiga)-1):
      for j in range(i+1,len(rotAntiga)-1):
        nRota = optSwap(rotAntiga, i, j)
        newd = totaldist(nRota)
        if newd < bestd:
          print("Nova dist Calculado por 2-Opt:",newd)
          rotAntiga = nRota
          bestd = newd
          improved = True
          break
      if improved:
        break
  return rotAntiga
starsA = list(range(0,100))#estrelas Inesploradas (em Aberto)
starsF = [0]# estrelas exploradas


#Algoritmo Guloso normal
#Pega primeira posi  o
last = starsA.pop(0)
#Enquanto existirem estrelas para explorar
while starsA:
  #print("===\n")
  menor = getMenor(last,starsA)
  print("menor opcao local:")
  print(menor)
  #print("[",calcdist(last , menor[0]),"]")
  last = menor[0]
  starsA.remove(menor[0])
  starsF.append(menor[0])
#Volta pro sol
starsF.append(0)

#Algoritmo 2opt
starsF = twOpt(starsF)


print("Caminho escolhido: ",starsF)

#calcula dist ncia percorrida pelo algoritmo
dist = totaldist(starsF)
print("Dist ncia do algoritmo",dist)

```

```
91 #Salva caminho em Arquivo
92 file = open('2opt.txt','w')
93 for i in starsF:
94   file.write(str(i)+" ")
95 file.close()
96 #Criar arquivo
```

Por fim, o código para a visualização 3D. A princípio, são importadas as bibliotecas **bpy** e **mesh**, utilizadas para executar funções no Blender. Em seguida, é importado o **Path** do módulo **pathlib**. Ele foi necessário para modificar a função de abertura de arquivo, visto que o código é executado diretamente na interface do Blender. Foi necessário encontrar o caminho do arquivo de código utilizando a função **bpy.context.space_data.text.filepath** para abrir o arquivo com as posições das estrelas.

Com isso feito, o código segue as mesmas lógicas anteriores. As posições foram salvas no vetor **pos** e todo o processo do Algoritmo guloso é repetido até a linha 81. A partir daí, começa a ser executada a lógica para visualização. Os comandos nas linhas 85 e 86 são usados para selecionar todos os objetos ativos e deletá-los, respectivamente. Em seguida, a variável **colorChange** é criada para definir o tom de verde que influenciará nos objetos.

O arquivo **Guloso.txt** é aberto/criado como antes, e a variável auxiliar **c** é criada para contador. Então, um **for** passa por todas as posições do caminho **StarsF**. Os dados são escritos no arquivo (a variável **ind** poderia ser simplesmente substituída por **i**, mas isso foi uma mudança ainda não atualizada em todo o código).

Um **if** verifica se não estamos na primeira posição para iniciar a criação do ambiente. O comando na linha 103 cria um objeto do tipo plano, que é salvo na variável **obj**. Na linha 108, entramos no modo de edição e, na linha 108, deletamos todos os vértices do plano. Das linhas 111 a 118, as funções são preparadas para adicionar novos vértices, e o índice da estrela anterior à atual é obtido (no caso de a estrela ser o sol, ele deve pegar a penúltima do caminho, já que voltamos para lá no final da viagem). Em seguida, o nome do objeto é modificado para identificar o caminho que ele está tomando.

Nas linhas 121 e 122, são criados os vértices com base nas coordenadas das estrelas no vetor **pos**. Na linha 124, é criada uma aresta indo de um vértice ao outro. Os dados do objeto são atualizados com esse **mesh**, e saímos do modo de edição na linha 127. Em seguida, convertemos esse objeto de aresta em uma curva, e a espessura da linha é definida como 0.1.

Na linha 134, é criado um novo material atribuído ao objeto. Logo em seguida, o material utiliza nodes e recebe uma cor de brilho com base na sua posição no vetor **starsF**. Por fim, na linha 142, o material define sua força de brilho como 250.

Agora, para criar uma animação em que as linhas do caminho aparecem na ordem correta, o contador auxiliar é multiplicado por 10, e **frames** são criados para as propriedades **hide_viewport** e **hide_render**. Quando essas propriedades são verdadeiras, o objeto é ocultado no Blender e na renderização. Os **frames** para as arestas desaparecerem ficam no início da timeline e 12 **frames** antes da variável **inda**. Já os **frames** para as arestas aparecerem são definidos da linha 152 a 155, na posição **inda**.

Por último, é necessário criar esferas para representar as estrelas. De maneira semelhante às arestas, elas são criadas na linha 158 e usam a posição do **inda** em **pos**. Recebem o nome de seu índice, e um novo material é criado. Esse material

também usa nodes para as estrelas brilharem com sua cor relativa à ordem em que são visitadas. No entanto, as estrelas aparecem o tempo todo, logo, a animação é criada para que elas brilhem com muita intensidade quando o algoritmo passar por elas.

Assim, nas linhas 168 a 170, são adicionados **frames** para o brilho 50, 12 **frames** antes e depois do **inda**. As linhas 172 e 173 definem o **frame inda** para o brilho 10000. O contador auxiliar **c** é incrementado, e ao sair do **for**, o arquivo das posições é fechado.

```
#(Observa  o: se o blender n o estiver encontrando o caminho
   pode ser necess rio salvar o arquivo novamente na pasta
   desejada pois ao mover o projeto ele pode estar tentando usar
   o caminho do arquivo anterior)

import bpy, bmesh
import numpy as np

from pathlib import Path

#path = some_object.filepath # as declared by props
path = str(bpy.context.space_data.text.filepath)


#pega caminho de si mesmo
selfp = Path(bpy.path.abspath(path))

#Pega diret rio da pasta
pathdir = Path(selfp).resolve().parent

print(pathdir)
#Pega arquivo posStars.txt
pfile = str(pathdir) + "\posStars.txt"

#Pega caminho do arquivo

    #Abre arquivo
with open(pfile) as file:
        #) # le texto do arquivo
        with open(pfile) as f:
            pos = []
            for line in f: # read rest of lines
                pos.append([float(x) for x in line.split()])


#Fun  o calcular dist ncia
def calcdist(start, end):
  return np.linalg.norm([x1-x2 for x1, x2 in zip(pos[start] , pos
   [end])])

#Fun  o calcular dist ncia
def getMenor(start,Astar,ig = []):
  ig.append(start)#n  o tentar voltar para o come o

  menor = Astar[0]
  for i in Astar:
    dm = 0
    dn = 0

    if i not in ig:
      dm += calcdist(start,menor)
```

```
      dn += calcdist(start,i)
      if dn < dm:
        menor = i
        dm = dn
  return [menor, dm]


starsA = list(range(0,100))#estrelas Inesploradas (em Aberto)
starsF = [0]# estrelas exploradas

#ALGORITMO GULOSO
#Pega primeira posi  o
last = starsA.pop(0)
#Enquanto existirem estrelas para explorar
while starsA:
  #print("===\n")
  menor = getMenor(last,starsA)
  print("menor op   o local:")
  print(menor)
  #print("[",calcdist(last , menor[0]),"]")
  last = menor[0]
  starsA.remove(menor[0])
  starsF.append(menor[0])
#Volta pro sol
starsF.append(0)
print("Caminho escolhido: ",starsF)

#calcula dist ncia percorrida pelo algoritmo
dist = 0
x = starsF[0]
for i in starsF:
  dist += calcdist(x, i)
  x = i
print("Dist ncia do algoritmo",dist)
#============================================


#Seleciona objetos do blender e deleta eles
bpy.ops.object.select_all(action='SELECT')
bpy.ops.object.delete(use_global=False, confirm=False)



colorChange = 1

file = open('Guloso.txt','w')
c = 0
for i in starsF:
  #Salav caminho escolhido no arquivo
  file.write(str(i)+" ")

  #Pega index do objeto
  ind = i

  if ind > 0 or c>0:
      bpy.ops.mesh.primitive_plane_add(enter_editmode=False,
    align='WORLD', location=(0, 0, 0), scale=(1, 1, 1))
      obj = bpy.context.object


```

```
107       #Entra edimode
108       bpy.ops.object.editmode_toggle()
109       bpy.ops.mesh.delete(type='VERT')
110
111       me = obj.data
112       bm = bmesh.from_edit_mesh(me)
113
114       if i != 0:
115         antes = starsF[starsF.index(i)-1]
116       else:
117         antes = starsF[starsF.index(i)-2]
118       print(antes," -> ", i)
119
120       obj.name =str(ind)+ " : "+str(antes)+" -> "+str(i)
121       v1 = bm.verts.new((pos[antes][0],pos[antes][1],pos[antes
      ][2]))
122       v2 = bm.verts.new((pos[ind][0],pos[ind][1],pos[ind][2]))
123
124       bm.edges.new((v1, v2))
125       #sai edimode
126       bmesh.update_edit_mesh(obj.data)
127       bpy.ops.object.editmode_toggle()
128
129       #Converte pra curva e seta atributos para ser visivel
130       bpy.ops.object.convert(target='CURVE')
131       bpy.context.object.data.bevel_depth = 0.1
132
133       #Cor
134       matr = bpy.data.materials.new("trail"+str(i))
135       bpy.context.object.active_material = matr
136
137       bpy.context.object.active_material.use_nodes = True
138       bpy.context.object.active_material.node_tree.nodes["
      Principled BSDF"].inputs[19].default_value = (1,colorChange*
      starsF.index(ind)/(len(starsF)),0,1)
139       if ind == 0:
140            bpy.context.object.active_material.node_tree.nodes["
      Principled BSDF"].inputs[19].default_value = (1,colorChange
      *101/(len(starsF)),0,1)
141
142       bpy.context.object.active_material.node_tree.nodes["
      Principled BSDF"].inputs[20].default_value = 250
143
144       inda = (c*10)
145       print(inda)
146       bpy.context.object.hide_viewport = True
147       bpy.context.object.hide_render = True
148       bpy.context.object.keyframe_insert('hide_viewport',frame =
      0)
149       bpy.context.object.keyframe_insert('hide_viewport',frame =
      inda+12)
150       bpy.context.object.keyframe_insert('hide_render',frame = 0)
151       bpy.context.object.keyframe_insert('hide_render',frame =
      inda+12)
152       bpy.context.object.hide_viewport = False
153       bpy.context.object.hide_render = False
154       bpy.context.object.keyframe_insert('hide_viewport',frame =
      inda)
155       bpy.context.object.keyframe_insert('hide_render',frame =
      inda)
```

```
156
157         #Cria esfera
158         bpy.ops.mesh.primitive_ico_sphere_add(radius=1,
    enter_editmode=False, align='WORLD', location=(pos[ind][0],
    pos[ind][1], pos[ind][2]), scale=(0.2, 0.2, 0.2))
159         bpy.context.object.name = str(i)#Muda nome da esfera
160
161         #Cria material para estrela
162         mt = bpy.data.materials.new("star"+str(i))
163         bpy.context.object.data.materials.append(mt)#Associa
    material a estrela
164         #Criar anima  o da luz da estrela
165         bpy.context.object.active_material.use_nodes = True
166         bpy.context.object.active_material.node_tree.nodes["
    Principled BSDF"].inputs[19].default_value = (1,colorChange*
    starsF.index(ind)/(len(starsF)),0,1)
167
168         bpy.context.object.active_material.node_tree.nodes["
    Principled BSDF"].inputs[20].default_value = 50
169         bpy.context.object.active_material.node_tree.nodes["
    Principled BSDF"].inputs[20].keyframe_insert(data_path="
    default_value", frame=inda-12)
170         bpy.context.object.active_material.node_tree.nodes["
    Principled BSDF"].inputs[20].keyframe_insert(data_path="
    default_value", frame=inda+12)
171
172         bpy.context.object.active_material.node_tree.nodes["
    Principled BSDF"].inputs[20].default_value = 10000
173         bpy.context.object.active_material.node_tree.nodes["
    Principled BSDF"].inputs[20].keyframe_insert(data_path="
    default_value", frame=inda)
174
175         c+=1
176
177
178 #Fecha Arquivo
179 file.close()
```

6 Resultados

Como esperado, o resultado obtido pelo algoritmo guloso não é globalmente ótimo, já que sempre escolhe um caminho localmente ótimo e não leva em consideração o resultado global. Porém, é um resultado satisfatório e nos permite analisar facilmente como o algoritmo se comporta e nos mostra o quão importante é levar em consideração as etapas seguintes na resolução do problema de caixeiro viajante, não apenas focar no próximo passo.

O caminho escolhido pelo algoritmo guloso foi o seguinte: [0, 3, 1, 2, 4, 7, 23, 41, 16, 9, 28, 40, 38, 35, 66, 75, 91, 86, 46, 64, 44, 77, 82, 69, 63, 55, 45, 31, 27, 20, 59, 74, 76, 68, 56, 12, 11, 24, 17, 43, 84, 70, 52, 51, 81, 60, 78, 39, 37, 5, 10, 34, 48, 50, 85, 79, 80, 65, 62, 36, 25, 29, 67, 89, 33, 53, 49, 94, 92, 88, 15, 21, 26, 6, 22, 8, 19, 18, 30, 99, 95, 98, 42, 54, 57, 72, 61, 73, 87, 97, 13, 14, 32, 58, 93, 83, 71, 96, 90, 47, 0] (o percurso todo está representado na Figura 1), com uma distância percorrida de 2108,452795703646.

Já no caminho do algoritmo 2-opt tivemos uma leve melhoria no desempenho. Como se trata de um algoritmo que o objetivo é otimizar uma solução existente,

buscando uma rota mais curta e eficiente, já era de se esperar que o caminho fosse otimizado com o auxílio do algoritmo 2-opt.

O caminho otimizado pelo algoritmo 2-opt foi o seguinte: [0, 3, 1, 2, 4, 7, 29, 44, 64, 46, 86, 91, 75, 66, 99, 98, 95, 87, 42, 54, 57, 72, 61, 22, 73, 97, 33, 53, 49, 94, 92, 88, 34, 48, 50, 85, 79, 80, 65, 62, 36, 25, 67, 89, 82, 77, 69, 63, 55, 45, 31, 20, 27, 59, 74, 76, 68, 56, 23, 41, 16, 9, 28, 35, 38, 40, 30, 18, 19, 8, 6, 26, 21, 15, 10, 5, 37, 39, 60, 78, 81, 51, 52, 70, 84, 43, 17, 24, 12, 11, 90, 96, 71, 83, 93, 58, 14, 13, 32, 47, 0] (o percurso todo está representado na Figura 2), com uma distância percorrida de 1919,4768785590331 (aproximadamente 8,96% de otimização).

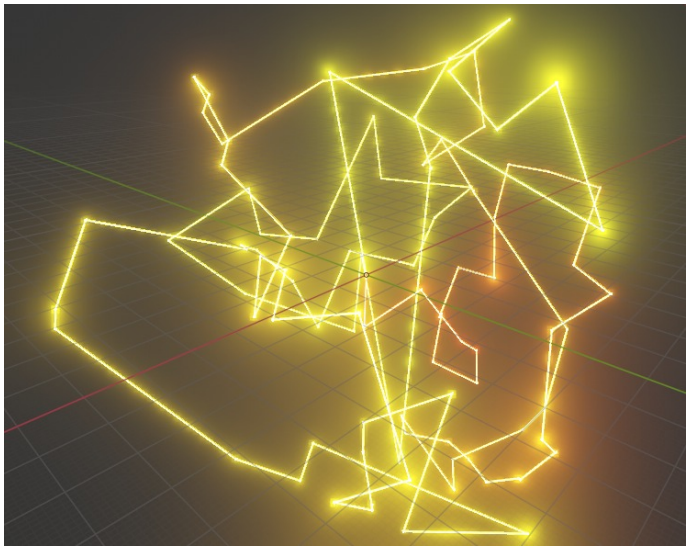

Figura 1: Mapa 3D do caminho percorrido pelo algoritmo guloso

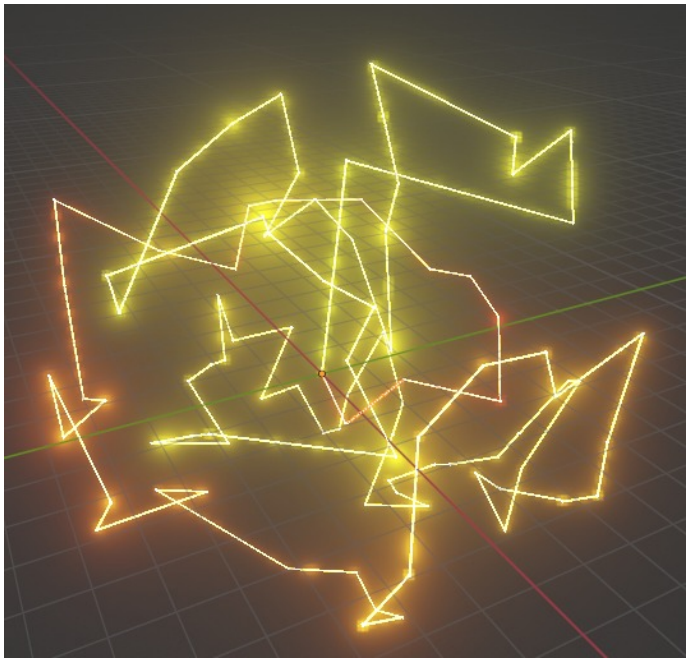

Figura 2: Mapa 3D do caminho percorrido pelo algoritmo 2-opt

7 Conclusão

Com base nos resultados obtidos, concluímos que o algoritmo guloso não produz uma solução globalmente ótima para o problema do caixeiro viajante com as 100 estrelas. No entanto, ele fornece uma solução satisfatória, considerando apenas escolhas localmente ótimas em cada etapa.

Por outro lado, ao aplicar o algoritmo 2-opt à solução inicial do algoritmo guloso, observamos uma melhoria significativa. O caminho otimizado pelo algoritmo 2-opt apresentou uma distância aproximadamente 8,96% menor em relação à solução inicial. Essa otimização ocorre devido à capacidade do algoritmo 2-opt de explorar e realizar trocas locais entre arestas, buscando reduzir a distância total percorrida na rota. Portanto, o algoritmo 2-opt demonstrou ser eficaz em melhorar a solução inicial do algoritmo guloso.

No entanto, é importante ressaltar que o algoritmo 2-opt também pode não garantir a solução globalmente ótima, pois está limitado a trocas locais entre pares de arestas. Para obter uma solução ainda mais otimizada, podem ser exploradas outras técnicas e algoritmos mais avançados.

Em suma, a combinação do algoritmo guloso seguido pelo algoritmo 2-opt demonstrou ser uma abordagem promissora para resolver o problema do caixeiro viajante com as 100 estrelas. As soluções obtidas fornecem insights valiosos sobre a importância de considerar o contexto global e a possibilidade de otimizações locais para alcançar resultados mais eficientes.

8 Referências

Espressif. Disponível em <https://www.espressif.com/en/products/socs/esp8266>

PEREIRA, Fábio. Tecnologia ARM: microcontroladores de 32 bits. 1. ed. São Paulo, SP: Érica, 2007. 448 p. ISBN 9788536501703.