CSI 466 - Teoria dos Grafos
Problema do Caixeiro Viajante

Lucas Monteiro Martino Cota[1]

[1]Instituto de Ciências Exatas e Aplicadas (ICEA)
Universidade Federal de Ouro Preto (UFOP) - João Monlevade – MG – Brasil

lucas.cota@aluno.ufop.edu.br

Abstract. *One of the most studied problems in computation is certainly the Traveling Salesman Problem (TSP), this is due to the fact that it can be applied in several cases of logistics and planning, such as in situations of school bus routing and scheduling of work teams. Although this problem is considered NP-complex, there are some strategies to find satisfactory solutions and this article will focus justly on analyzing two of these solutions, in relation to the execution time and cost of the best route found. The algorithms to be analyzed are the Brute Force algorithm and the Nearest Neighbor algorithm with refinement.*

Resumo. *Um dos problemas mais estudados da computação certamente é o Problema do Caixeiro Viajante (PCV), isso se deve ao fato de poder ser aplicado em diversos casos de logística e planejamento, como em situações de roteamento de ônibus escolares e agendamento de equipes de trabalho. Apesar de este problema ser considerado do tipo NP-complexo existem algumas estratégias para encontrar soluções satisfatórias e este artigo vai focar justamente em analisar duas destas soluções, em relação ao tempo de execução e ao custo da melhor rota encontrada. Os algoritmos a serem analisados são o algoritmo de Força Bruta e o algoritmo do Vizinho Mais Próximo com refinamento.*

1. Introdução

A Teoria do Grafos é uma disciplina que aborda o estudo das relações entre objetos de um determinado conjunto através de grafos G(V, E, w). Ao começar o estudo de grafos você percebe que vários problemas da vida real podem ser modelados em grafos, tornando mais fácil o seu entendimento e a sua resolução.

Do mesmo modo que é simples transformar um problema real em um grafo, também é simples representar este grafo computacionalmente e se, um problema real em forma de grafo já era mais fácil de se lidar, com a ajuda de um computador fica ainda mais. Hoje em dia existem diversos algoritmos que ajudam na solução dos mais variados desafios que podem vir a aparecer, mas também é verdade que ainda existem aqueles problemas que ainda não possuem uma solução exata.

Estes problemas que não possuem uma solução exata em tempo polinomial são chamados de ”NP-Complexo (ou, NP-Difícil)”. Nesta classe estão diversos problemas e entre eles o Problema do Caixeiro Viajante (PCV), que consiste em encontrar o ciclo hamiltoniano de custo mínimo em um grafo ponderado [FONSECA 2021].

Ainda que o Problema do Caixeiro Viajante seja considerado NP-complexo, existem diversas estratégias para encontrar uma boa solução e o objetivo deste artigo é analisar duas dessas formas. Serão analisadas duas abordagens diferentes, uma abordagem heurística que pode não encontrar a melhor rota possível mas executa em um tempo muito pequeno mesmo para grafos enormes e uma abordagem exata que encontra a melhor rota possível mas em um tempo não muito pequeno.

A abordagem heurística vai ser por meio do algoritmo guloso do Vizinho Mais Próximo e a abordagem exata vai ser por meio do algoritmo de Força Bruta e ao final dos experimentos será possível chegar a conclusão de qual dos dois tipos de abordagem é mais vantajoso, isto é, levando em conta também a necessidade de quem vai utilizá-los, se pode ou não demorar muito tempo e se é necessário sempre a melhor rota possível.

2. Algoritmos

Nesta seção as funções do código utilizado no artigo vão ser detalhadas e explicadas para um melhor entendimento do programa e de como ele deve ser utilizado. Como as funções de preencher uma matriz e escrever em um arquivo de saída são mais simples elas não serão explicadas nesta seção e estão armazenada em um repositório no GitHub [COTA 2021] juntamente com o código fonte completo.

As funções explicadas serão os algoritmos de resolução do Problema do Caixeiro Viajante, sendo eles o algoritmo de Força Bruta e o algoritmo do Vizinho Mais Próximo com refinamento k-opt, mas antes deles vem a **main** e a sua explicação logo após o código abaixo.

Algoritmo 1. main

```
nomeArquivo = input("\nInforme o grafo: ")
arquivo = open(nomeArquivo, 'r')

cabecalho = arquivo.readline()
cabecalho = cabecalho.split()
numVertices = int(cabecalho[0])
numArestas = float(cabecalho[1])

matriz = []
matriz.append(arquivo.readlines())
matAdj = [[0 for i in range(numVertices)]for i in range(numVertices)]
matAdj = preencheMatrizAdjacencia(matriz, matAdj)

print("\nAlgoritmos disponiveis:\n\n1- Vizinho Mais Proximo\n2- Forca
    Bruta")
opcao = int(input("\nDigite o numero do algoritmo que voce deseja
    utilizar: "))

if opcao == 1 or opcao == 2:
    tempo = int(input("\nTempo limite (s): "))
    custo = [[] for i in range(2)]
    if opcao == 1:
        opt = int(input("Digite o nivel de refinamento: "))
        tempoExecucao = [[] for i in range(2)]
        refinamento_k_opt(matAdj, tempo, opt)
    else:
```

```
25          forcaBruta(matAdj, tempo)
26  else:
27      print("\nOpcao invalida ! Programa encerrado !")
28
29  arquivo.close()
```

Na primeira linha do código acima é feita a leitura do nome do arquivo que será aberto e nele a primeira linha tem que ser necessariamente o número de vértices e o número de arestas. Depois que o número de vértices e arestas são salvos a matriz de adjacências é criadas de acordo com o grafo do arquivo.

Após a criação da matriz é exibido um menu no console para o usuário escolher entre o algoritmo do Vizinho Mais Próximo e de Força Bruta, caso escolha o primeiro o tempo limite de execução do programa em segundos é perguntado e também o nível de refinamento que se deseja, caso seja o algoritmo de Força Bruta somente o tempo de execução é perguntado. Em seguida, ou se for escolhida um opção inválida, o arquivo é fechado é o programa encerrado.

O código abaixo é o do Vizinho Mais Próximo, que recebe a matriz com o peso das arestas, o tempo máximo de execução e o índice (0 se for a primeira execução, 1 se for a execução com refinamento). A lista **Q** armazena todos os vértices a serem processados e o algoritmo executa até ela ficar vazia, adicionando a cada iteração um par de vértices na lista do caminho final **C** e removendo o vértice adicionado por último de **Q**, esta função também calcula o custo do caminho percorrido e o tempo de execução e os salva em duas variáveis globais. Outras maneiras de o algoritmo encerrar sem ser com a lista **Q** vazia é se o tempo limite já tiver sido superado ou se não existe um caminho para o nivel de refinamento escolhido.

Algoritmo 2. Vizinho Mais Próximo

```
1   def vizinhoMaisProximo(G, tempo, indice):
2       inicio = time.time()
3
4       u = 0   # Vertice atual
5       C = []  # Caminho final
6       Q = [i for i in range(len(G))]  # Lista de vertices a serem
            processados
7       Q.remove(u)
8       custo[indice] = 0
9
10      while len(Q) != 0:  # Enquanto houver vertices a serem processados
11          min = float("inf")
12          v = -1
13
14          for i in Q:     # Procura pelo vizinho mais proximo
15              if min > G[u][i] and G[u][i] != 0:
16                  min = G[u][i]
17                  v = i
18          if v == -1:
19              print("\nNao foi possivel concluir um caminho com este
                    nivel de refinamento !")
20              return []
21
22          custo[indice] += min
```

```
23
24          C.append((u, v))      # Adiciona o caminho do vizinho mais
                proxima a lista
25          Q.remove(v)           # Remove o vertice dos vertices nao
                processados
26          u = v                 # Atualiza para o proximo vertice
27
28          aux = time.time()
29          if (aux - inicio) > tempo:
30              print("\nNao foi possivel achar a rota pelo metodo do
                    Vizinho Mais Proximo no limite de tempo estabelecido !"
                    )
31              return []
32
33      custo[indice] += G[u][0]
34      C.append((u, 0))      # Adiciona o ultimo vertice para fechar o ciclo
35      fim = time.time()
36
37      tempoExecucao[indice] = fim - inicio      # Tempo de execucao do
            algoritmo
38
39      return C      # Retorna o caminho percorrido
```

Para o refinamento do código do Vizinho Mais Próximo é utilizada a função abaixo, ela executa o algoritmo do Vizinho Mais Próximo uma vez e logo depois remove as **k** maiores arestas, estas são salvas na lista **arestasExcluidas**. As arestas que vão ser excluídas recebem 0 na matriz com o peso das arestas para não poderem ser acessadas e sem estas arestas o algoritmo do Vizinho Mais Próximo é executado novamente e logo depois é exibido se o algoritmo teve um caminho melhor com ou sem refinamento, o custo e o tempo de execução. O melhor caminho encontrado é escrito em um arquivo de saída com o custo dele.

Algoritmo 3. Refinamento k-opt

```
1  def refinamento_k_opt(G, tempo, opt):
2      C = vizinhoMaisProximo(G, tempo, 0)
3
4      tamanhoCaminho = [[] for i in range(len(C))]
5      arestasExcluidas = [[] for i in range(opt)]
6
7      j = 0
8
9      for i in C:
10         tamanhoCaminho[j] = G[i[0]][i[1]]    # Pesos do caminho
               percorrido
11         j += 1
12
13     for i in range(opt):
14         arestasExcluidas[i] = max(tamanhoCaminho)
15         indice = tamanhoCaminho.index(max(tamanhoCaminho))    # Indice
               do maior caminho percorrido
16         tamanhoCaminho[indice] = float('-inf') # O maior caminho recebe
               o valor "-infinito"
17         G[C[indice][0]][C[indice][1]] = 0    # As arestas excluidas
               recebem 0
```

```
18          G[C[indice][1]][C[indice][0]] = 0
19
20      C2 = vizinhoMaisProximo(G, tempo, 1)
21      if custo[1] > custo[0] or not C2:
22          print("\nO caminho sem refinamento tem um custo menor !")
23          print("Custo: {:.2f}". format(custo[0]))
24          print("Tempo:", tempoExecucao[0], "s\n")
25          escreverArquivoSaida(C, 0)
26      else:
27          print("\nO caminho com refinamento tem um custo menor !")
28          print("Arestas excluidas:", arestasExcluidas)
29          print("Custo: {:.2f}". format(custo[1]))
30          print("Tempo:", tempoExecucao[1], "s\n")
31          escreverArquivoSaida(C2, 1)
```

Já o algoritmo de Força Bruta é exibido a seguir, ele possui uma lista auxiliar **C** que é responsável por armazenar todos caminhos possíveis e este algoritmo verifica um por um para ver qual tem o menor custo, o caminho com menor custo é salvo em **C_best** e o custo deste caminho em **cost**. Caso o tempo limite seja extrapolado o algoritmo retorna o melhor caminho encontrado até o momento, este melhor caminho é salvo em um arquivo de saída juntamente com o seu custo.

Algoritmo 4. Força Bruta

```
1  def forcaBruta(G, tempo):
2      inicio = time.time()
3      cost = float("inf")                # Custo do melhor caminho
4      C_best = []                        # Melhor caminho
5      C = [i for i in range(len(G))]     # Lista de caminho auxiliar
6
7      perm_iterator = itertools.permutations(C)   # Realiza as
           permutacoes da lista auxiliar
8
9      for C in perm_iterator:                     # Percorre as listas
           geradas pela permutacao
10         C = list(C)                             # Convertendo a tupla em
               lista
11         C.append(C[0])                          # Adicionando o primeiro
               elemento para fechar o ciclo
12         custo_atual = 0
13         i = 0
14         while (i + 1) < len(C):
15             custo_atual += G[C[i]][C[i+1]]      # Calcula o custo da
                   permutacao atual
16             i += 1
17         if cost > custo_atual:                  # Verifica se o caminho atual
               e menor que o menor caminho encontrado
18             cost = custo_atual
19             C_best = C
20         aux = time.time()
21
22         if (aux - inicio) > tempo:              # Retorna a melhor rota ate o
               tempo maximo estabelecido
23             fim = time.time()
24             print("\nMelhor rota em ate", tempo, "segundos !")
```

```
25              print("Custo: ", cost)
26              print("Tempo:", fim - inicio, "s")
27              escreverArquivoSaidaForcaBruta(C_best, cost)
28              return C_best
29
30      fim = time.time()
31      print("Custo: ", cost)
32      print("Tempo:", fim - inicio, "s")
33      escreverArquivoSaidaForcaBruta(C_best, cost)
34
35      return C_best
```

3. Experimentos e Resultados

Nesta seção os resultados obtidos através dos experimentos serão exibidos, analisados e comentados. Os testes foram feitos sobre os cinco grafos disponibilizados com um limite de tempo de 60 segundos por um notebook com processador Intel Core i5-8265U com 16GB de memória RAM e quatro núcleos operando a 1.60GHz cada.

3.1. Vizinho Mais Próximo

Os testes sobre o algoritmo do Vizinho Mais Próximo foram realizados primeiramente sem o refinamento e os resultados obtidos são demonstrados na Tabela 1, na Tabela 2 são exibidos os resultados que um refinamento de 3-opt causou. Mesmo podendo ser feito um refinamento de mais arestas, a escolha pelo refinamento de 3-opt foi por causa de que a exclusão de muitas arestas pode acabar ficando sem caminho e em muitas ocasiões o custo da melhor rota acaba não sendo melhor que o custo da rota sem refinamento.

Com o refinamento por 3-opt foram atingidas rotas melhores em três casos e os tempos de execução além de ficarem muito baixos, são bem próximos do algoritmo sem refinamento.

Tabela 1. Vizinho Mais Próxima: Sem Refinamento

Arquivo	Custo da melhor rota	Tempo de execução
a280.txt	3148.42	0.0047s
ali535.txt	2918.38	0.0129s
ch130.txt	7575.28	0.0009s
fl1577.txt	27343.87	0.0873s
gr666.txt	4110.90	0.0239s

Tabela 2. Vizinho Mais Próxima: Refinamento 3-opt

Arquivo	Custo da melhor rota	Tempo de execução
a280.txt	3255.59	0.0029s
ali535.txt	2907.69	0.0109s
ch130.txt	7165.01	0.0010s
fl1577.txt	27517.89	0.1226s
gr666.txt	4023.13	0.0179s

3.2. Força Bruta

Os resultados alcançados sobre os testes no algoritmo de Força Bruta são exibidos na Tabela 3. Em nenhum dos grafos o algoritmo foi capaz de achar a melhor rota em menos de 60 segundos e os resultados em termos da melhor rota foram bem distintos, com o algoritmo sendo capaz de encontrar uma melhor rota para o grafo a280.txt do que o Vizinho Mais Próximo com refinamento 3-opt e achando rotas muito maiores nos outros, especialmente nos grafos ali535.txt e ch130.txt.

Tabela 3. Força Bruta

Arquivo	Custo da melhor rota	Tempo de execução
a280.txt	2818.62	60.01s
ali535.txt	36807.39	60.00s
ch130.txt	46583.32	60.00s
fl1577.txt	51065.31	60.00s
gr666.txt	5427.12	60.00s

4. Conclusão

Chegando ao final do trabalho foi possível entender melhor o que é e como funciona o Problema do Caixeiro Viajante e que apesar de ser um problema NP-complexo ele possui alguns métodos de resolução que oferecem boas soluções.

Dois destes métodos de solução foram explorados neste artigo, o método do Vizinho Mais Próximo com refinamento em que foi visto que seria o ideal para casos que precisem de boas soluções e em tempos muito pequenos e também o método da Força Bruta, um método que acaba sendo aleatório para limite de tempos pequenos, podendo oferecer uma solução boa ou ruim, mas ele seria o ideal para casos em que fosse necessário a melhor rota e não houvesse limite de tempo.

Referências

FONSECA, George Henrique Godim da. Notas de aula, 2021. Universidade Federal de Ouro Preto.

COTA, Lucas Monteiro Martino. GitHub, 2021. Busca em Largura e em Profundidade. Disponível em: https://github.com/lucasmmcota/travelingSalesmanProblem. Acesso em: 24 de julho de 2021.