UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS DE CRATEÚS

Relatório sobre o trabalho final de Algoritmos em Grafos

Deyvid Nascimento de Sousa
Jorge Roniel de Paula Souza
Lúcio Cauper Freitas

Crateús

2025

1. Introdução

No mundo da ciência da computação, vários problemas foram surgindo através dos anos, muitos com soluções simples, e outros nem tanto, com soluções mais complexas, já outros até os dias atuais permanecem sem solução, principalmente em tempo polinomial, os famosos NP-Difíceis, dentre eles temos o Problema do Caixeiro Viajante (PCV). Este problema é de otimização NP-difícil e consiste na procura de um ciclo que possua a menor distância, começando a partir de uma cidade qualquer, entre várias, e visitando cada cidade uma vez e regressando a cidade inicial.

Neste trabalho, utilizaremos de uma variação do PCV, o PCV-métrico, onde as distâncias entre duas cidades seguem a desigualdade triangular. Para uma solução aproximada, o algoritmo de Christofides será implementado, com ele, teremos uma aproximação da solução de 1,5, que é a segunda aproximação com um comprimento mais próxima de uma solução ótima para o PCV-métrico.

Diante de tal contextualização, este presente trabalho tem como objetivo, apresentar uma solução aproximada para o problema do Caixeiro viajante, usando grafos completos com pesos, satisfazendo a desigualdade triangular, utilizando o Algoritmo de Christofides.

1.1 Objetivos e organização

Este trabalho tem como objetivo demonstrar uma solução aproximada do problema do caixeiro viajante, com as devidas características inerentes necessárias, utilizando o algoritmo de Christofides.

Este relatório está organizado em 4 capítulos, sendo este primeiro relativo à introdução. O segundo capítulo será uma pequena contextualização do problema do caixeiro viajante e do algoritmo de Christofides. No terceiro capítulo teremos a metodologia deste trabalho. No quarto capítulo será descrito resultados e discussões.

Uma observação que não faz parte do trabalho porém inerente a visualização é ler este pdf em uma tela suficientemente grande para não ter problemas com o texto.

2. Contextualização

2.1 Problema do Caixeiro Viajante

Desde 1800, problemas relacionados ao problema do caixeiro viajante começaram a surgir, sendo desenvolvidos por dois matemáticos, o escocês William Rowan Hamilton e o britânico Thomas Penyngton Kerkman. Porém, só em 1930 a forma geral do PCV começou a ser estudada em Harvard e Viena e o problema começou a ficar globalmente conhecido em 1950.

O Problema do Caixeiro Viajante, ou *Travelling Salesman Problem*, consiste no seguinte enunciado: "Um caixeiro viajante quer sair de sua cidade para vender suas mercadorias, passar por todas as cidades da sua região, sem repeti-las, e voltar para a sua cidade natal", ou seja, é um problema que visa determinar o menor caminho possível para percorrer uma série de cidades, visitando-as uma única vez, retornando à cidade de origem.

Contudo, esse problema pode ficar muito complexo ao aumentar o número de cidades, pois o tamanho do espaço de procura aumenta exponencialmente dependendo de N que é o número de cidades. Dessa forma, o problema é conhecido como NP-Difícil, já que não existe um algoritmo que, em tempo polinomial, possa encontrar uma solução ótima.

O PCV possui algumas variações, dentre elas existe o PCV-métrico, que possui uma restrição que é a desigualdade triangular, essa restrição impõe ao problema que a distância direta entre duas cidades é menor ou igual do que partir de uma cidade para outra por meio de uma cidade intermediária: $d(A,C) \leq d(A,B) + d(B,C)$. Com essa variação podemos implementar o algoritmo de Christofides que nos trás soluções aproximadas para o PCV-Métrico.

2.2 Algoritmo de Christofides

Publicado em 1976 por Nicos Christofides, o algoritmo de Christofides é um algoritmo feito para encontrar soluções aproximadas para o Problema do Caixeiro Viajante, na variação onde as distâncias são simétricas e obedecem a desigualdade triangular. Este algoritmo de aproximação garante uma solução aproximada de 1,5 $\left(\frac{3}{2}\right)$ da solução ótima.

Seu funcionamento se dá na seguinte forma:

1. Crie uma árvore geradora mínima T de G.
2. Seja O um conjunto com os vértices com grau ímpar de T.
3. Encontre o emparelhamento perfeito M no subgrafo induzido em G por O.
4. Combine M e T e forme um multigrafo H.
5. Forme um ciclo Euleriano em H.
6. Transforme o ciclo Euleriano em H para um ciclo Hamiltoniano pulando os vértices repetidos.

Neste trabalho, usaremos o algoritmo de Christofides composto de outro algoritmos em grafos para que seu funcionamento seja satisfatório.

3. Metodologia

Para a solução aproximada do problema, foi utilizado a linguagem de programação Python por sua popularidade e facilidade de análise, juntamente com a biblioteca *networkx*, na qual foi utilizada para o emparelhamento perfeito mínimo, *heapq*, que foi utilizada para manipular o heap mínimo em prim, e a biblioteca *os* utilizada para validação da existência do caminho do arquivo. Vale ressaltar que arestas com peso 0 fora da diagonal principal são tratadas como uma aresta válida e não como ausência de aresta.

Na estrutura do código, que está disponível no GitHub, tem-se um arquivo python chamado "*christofides.py*", no qual contém o código principal que será comentado posteriormente no item 3.1, além de outros 10 arquivos txt que contém descrições de grafos em forma de matriz de adjacência como especificado na descrição do trabalho e os arquivos txt de suas respectivas soluções.

3.1 Estrutura do código principal *christofides.py*

O arquivo *christofides.py* contém diversas funções relativas aos procedimentos a serem feitos pelo algoritmo de christofides, cada uma das funções serão discutidas nos tópicos seguintes juntamente com o código realizado para tal. A descrição das funções estão organizadas de forma respectivamente a forma na qual está escrita no código original.

3.1.1 leitura_arquivo_e_validacao

```
5: def leitura_arquivo_e_validacao(caminho: str):
6:     if not os.path.exists(caminho):
7:         print(f"Erro: O arquivo '{caminho}' nao foi encontrado. Por favor, verifique o caminho e o
nome do arquivo.")
8:         return None, ""
9:
10:     with open(caminho, 'r') as f:
11:         linhas = f.read().splitlines()
12:
13:     if not linhas:
14:         print(f"Erro: O arquivo '{caminho}' esta vazio.")
15:         return None, ""
16:
17:     try:
18:         n = int(linhas[0])
19:     except ValueError:
20:             print(f"Erro: A primeira linha do arquivo '{caminho}' deve ser um numero inteiro
(quantidade de vertices).")
21:         return None, ""
22:
23:     if n <= 0:
24:         print(f"Erro: O numero de vertices deve ser positivo e maior que zero.")
```

```
25:         return None, ""
26:
27:     matriz = []
28:
29:     if len(linhas) - 1 != n:
30:           print(f"Erro: O numero de linhas de dados no arquivo ({len(linhas) - 1}) nao corresponde
ao numero de vertices declarado ({n}).")
31:         return None, ""
32:
33:     for i in range(n):
34:         try:
35:             linha_valores = list(map(float, linhas[i + 1].split()))
36:         except ValueError:
37:               print(f"Erro: A linha {i + 2} do arquivo '{caminho}' contem valores nao numericos ou
mal formatados.")
38:             return None, ""
39:
40:         if len(linha_valores) != n:
41:              print(f"Erro: A linha {i + 2} da matriz nao tem o numero esperado de colunas ({n}). O
grafo nao e completo.")
42:             return None, ""
43:
44:         matriz.append(linha_valores)
45:
46:     for i in range(n):
47:         for j in range(n):
48:             if matriz[i][j] != matriz[j][i] or matriz[i][j] < 0:
49:                     print(f"Erro: O grafo nao e simetrico ou possui pesos negativos. Matriz[{i +
1}][{j + 1}] ({matriz[i][j]}) != Matriz[{j + 1}][{i + 1}] ({matriz[j][i]}) ou < 0.")
50:                 return None, ""
51:
52:     return matriz, "validado"
```

A função *leitura_arquivo_e_validacao* tem como objetivo fazer a leitura de um arquivo txt e depois a tradução para uma estrutura de matriz de adjacências, e durante essa tradução, ela realiza validações para garantir que o arquivo especifica um grafo válido para o algoritmo. O python abre o arquivo caso exista, lê e o chama de *f*, e armazena *f* em uma lista de linhas caso ele não seja vazio, a primeira linha é referente ao número de vértices, conforme exigido pelo trabalho, se o valor não for consistente à um inteiro válido, o grafo não é aceito, e por fim inicializa a matriz de adjacências, que vai ser o item a ser retornado pela função, também verificando se os valores estão bem formatados e se são valores esperados.

3.1.2 prim

```
54: def prim(grafo):
55:     vertices = list(range(len(grafo)))
56:     visitados = set()
57:     mst = {v: [] for v in vertices}
58:
59:     inicio = vertices[0]
60:     visitados.add(inicio)
```

```
61:
62:     heap = []
63:     for destino in vertices:
64:         if destino not in visitados:
65:
66:             heapq.heappush(heap, (grafo[inicio][destino], inicio, destino))
67:
68:     while heap:
69:         peso, origem, destino = heapq.heappop(heap)
70:         if destino not in visitados:
71:             visitados.add(destino)
72:             mst[origem].append((destino, peso))
73:             mst[destino].append((origem, peso))
74:             for aresta in vertices:
75:                     if aresta not in visitados:
76:                         heapq.heappush(heap, (grafo[destino][aresta], destino, aresta))
77:
78:     return mst
```

Nesta função, estamos realizando o algoritmo de prim, como o nome da função sugere, em resumo temos que o resultado dessa função é uma árvore geradora mínima, representada por um dicionário, onde as chaves são os vértices e cada valor das chaves representa a lista de adjacência (tuplas) daquele vértice. Das linhas 55 a 57, temos o armazenamento dos vértices em uma lista, inicialização de uma estrutura hashset onde vamos salvar os vértices já inseridos na árvore, e terminamos fazendo a inicialização da estrutura da árvore. Salvamos o primeiro vértice em uma variável para posteriormente usarmos as arestas referentes a ele para iniciar o algoritmo (vértice inicial). Nas linhas 62 a 66 fazemos um heap para colocar as arestas com menor peso do vértice inicial, preparando o início do algoritmo, evitando também pegar o valor da diagonal principal. Nas linhas 68 a 76 iniciamos o algoritmo do prim por meio de um loop while enquanto o heap não for vazio (enquanto existir arestas), removemos a aresta de menor valor do heap e armazenamos esses dados, se o vértice de destino da aresta não tiver sido adicionado (verificamos isso pelos elementos de visitados) significa que podemos adicionar essa aresta à árvore sem formar ciclos, então adicionamos a ida e a volta da aresta e adicionamos as arestas referentes à ela ao heap, quando o heap estiver vazio, sabemos que já visitamos adicionamos todos os vértices e verificamos as arestas de cada um deles, então temos uma árvore geradora mínima.

3.1.3 vertices_impares

```
80: def vertices_impares(arvore_geradora):
81:     return [v for v, adj in arvore_geradora.items() if len(adj) % 2 != 0]
```

Nesta função, temos apenas o retorno dos vértices de grau ímpar da árvore geradora mínima, iteramos pela dupla de elementos de um dicionário (chave: valor) e para cada chave (vértice) que possui uma lista de tamanho ímpar como valor, adicionamos o vértice à lista (vértice com grau ímpar).

3.1.4 grafos_auxiliar

```
83: def grafo_auxiliar(grafo, vertices):
84:     G = nx.Graph()
85:     for u in vertices:
86:         for v in vertices:
87:             peso = grafo[u][v]
88:             if u != v:
89:                 G.add_edge(u, v, weight=peso)
90:     return G
```

Na função grafos_auxiliar, ela tem como objetivo criar uma estrutura que seja aceita pela função do emparelhamento, posteriormente será retornado um grafo completo com os vértices de grau ímpar da árvore geradora mínima, vale ressaltar a importância de especificar o atributo peso com "weight" pois é através dele que a função de emparelhamento localiza o valor do peso utilizado em seu algoritmo.

3.1.5 ciclo_euleriano

```
92: def ciclo_euleriano(grafo):
93:     circuito = []
94:     stack = [0]
95:
96:     while stack:
97:         atual = stack[-1]
98:         if grafo[atual]:
99:             proxima = grafo[atual].pop(0)
100:             grafo[proxima[0]].remove((atual, proxima[1]))
101:             stack.append(proxima[0])
102:         else:
103:             circuito.append(stack.pop())
104:     return circuito
```

Essa função descreve como encontrar um ciclo euleriano, considerando um grafo com todos os vértices com grau par, usa-se uma estrutura de pilha para "percorrer" pelas arestas do grafo removendo-as no processo, quando encontra um ponto onde não é mais possível sair de um vértice por uma aresta, significa que a lista de adjacência daquele vértice está vazia, então desempilhamos o elemento do topo (o próprio vértice) e o adicionamos ao ciclo euleriano, ao desempilhar acabamos "voltando" no caminho percorrido anteriormente, com isso dizemos que estamos construindo o ciclo de trás para frente, quando a pilha estiver vazia, sabemos que encontramos um ciclo euleriano pois todos os vértices tinha grau par.

3.1.6 ciclo_hamiltoniano

```
106: def ciclo_hamiltoniano(circuito):
```

```
107:     ciclo = []
108:     for v in circuito:
109:         if v not in ciclo:
110:             ciclo.append(v)
111:     ciclo.append(ciclo[0])
112:     return ciclo
```

Nesta função temos o retorno de um ciclo hamiltoniano, com base na análise do ciclo euleriano previamente feito na função anterior, apenas iteramos pela lista de elementos do ciclo euleriano, criando outra lista, ignorando os elementos já lidos anteriormente.

3.1.7 custo_total

```
114: def custo_total(grafo, ciclo):
115:     total = 0.0
116:     for i in range(len(ciclo) - 1):
117:         origem = ciclo[i]
118:         destino = ciclo[i + 1]
119:         peso = grafo[origem][destino]
120:         total += peso
121:
122:     return total
```

Nesta função encontramos o custo total do ciclo hamiltoniano, como parâmetro, será recebido o grafo completo e o seu determinado ciclo hamiltoniano, será feita uma iteração pelos elementos da lista, de forma a ler o elemento atual da iteração e o próximo, de forma a utilizá-los para acessar o peso da aresta entre os 2 vértices, esse valor será somado à soma total.

3.1.8 christofides (função principal)

```
124: def christofides():
125:     grafo = None
126:     status = ""
127:
128:     while status != "validado":
129:         nome_arquivo = input("Por favor, digite o nome do arquivo .txt com a matriz de adjacencia
(ex: lin318.txt): ")
130:         grafo, status = leitura_arquivo_e_validacao(nome_arquivo)
131:
132:     print(f"\nGrafo '{nome_arquivo}' lido e validado com sucesso!")
133:     if grafo:
134:         mst = prim(grafo)
135:
136:         print("Arvore geradora minima: \n")
137:         print(mst)
138:         peso_mst = 0.0
139:         for v in mst.values():
140:             for (_, p) in v:
141:                 peso_mst += p
142:
```

```
143:         print(f"\nPeso total da Arvore geradora Minima: {peso_mst / 2}\n")
144:
145:         impares = vertices_impares(mst)
146:         aux = grafo_auxiliar(grafo, impares)
147:         emparelhamento = nx.algorithms.matching.min_weight_matching(aux)
148:
149:         for u, v in emparelhamento:
150:             peso = grafo[u][v]
151:             mst[u].append((v, peso))
152:             mst[v].append((u, peso))
153:
154:         circuito = ciclo_euleriano(mst)
155:         ciclo_final = ciclo_hamiltoniano(circuito)
156:         custo = custo_total(grafo, ciclo_final)
157:
158:         print("Solucao aproximada encontrada por Christofides: ")
159:         print(" -> ".join(map(str, ciclo_final)))
160:         print(f"\nPeso total da solucao aproximada: {custo}\n")
```

Na função principal temos a chamada de todas as outras funções citadas nos tópicos anteriores de forma procedural. A função inicia com uma variável *grafo* que vai receber o resultado da função *leitura_arquivo_e_validacao* (a matriz de adjacência), e com uma variável *status* que irá informar se o arquivo foi validado com sucesso, na estrutura de repetição posterior (while) será solicitado ao usuário o nome do arquivo, após uma série de validações, caso o arquivo especificado não seja validado, o usuário será solicitado novamente até que um arquivo válido seja informado. Na linha seguinte o grafo passa pelo algoritmo de prim e tem o resultado guardado na variável mst. Assim temos a árvore geradora mínima do grafo, tal árvore será impressa, logo após o peso da árvore vai ser calculado, e por se tratar de um grafo não direcionado, é preciso dividir o peso por 2. Na linha 145 encontramos os vértices de grau ímpar da árvore. Na linha 146 temos o subgrafo completo com apenas os vértices de grau ímpar da linha anterior, e em seguida é feito o emparelhamento. Com esse resultado, temos um multigrafo que é construído a partir da mst e do emparelhamento nas linhas 149 a 152, com isso garantimos grau par em todos os vértices. Após isso, encontramos o ciclo euleriano do multigrafo, um ciclo hamiltoniano e finalmente temos o custo total do ciclo hamiltoniano. E por fim temos um print, da solução aproximada e seu custo.

4. Resultados e discussões

Para realização de testes iniciais com o algoritmo de Christofides utilizamos as instâncias disponibilizadas pelo professor, os grafos si175 e bayg29. Posteriormente, após uma busca pela internet, encontramos um repositório com algumas outras instâncias de grafos para o problema do TSP, esse repositório pode ser encontrado em: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/. Vale ressaltar que esse repositório contém tanto instâncias válidas para o problema do Caixeiro Viajante quanto instâncias que não formam um plano métrico.

Tomamos o cuidado de selecionar e tratar apenas as instâncias com distâncias euclidianas, que podem ser modeladas adequadamente para o problema do trabalho, nos pontos seguintes desta seção será discutido os resultados do algoritmo ao usar como entrada a matriz de adjacência de cada um dos grafos utilizados para teste.

4.1 Testes realizados

4.1.1 bayg29

Peso total da Árvore Geradora Mínima: 1319.0
Peso total da solução aproximada: 1716.0
Peso total da solução ótima: 1610

Fator de Aproximação: 1716.0/1610 ≈ 1.066 vezes o ótimo

Mesmo peso de árvore mínima fornecido pelo professor

4.1.2 berlin52

Peso total da Árvore Geradora Mínima: 6078.0
Peso total da solução aproximada: 8595.0
Peso total da solução ótima: 7542

Fator de Aproximação: 8595.0/7542 ≈ 1.139 vezes o ótimo

4.1.3 bier127

Peso total da Árvore Geradora Mínima: 94706.0
Peso total da Solução aproximada: 133050.0
Peso total da Solução ótima: 118282

Fator de Aproximação: 133050.0/118282 ≈ 1.125 vezes o ótimo

4.1.4 d198

Peso total da Árvore Geradora Mínima: 11738.0
Peso total da Solução aproximada: 17263.0
Peso total da Solução ótima: 15780

Fator de Aproximação: 17263.0/15780 ≈ 1.094 vezes o ótimo

4.1.5 d657

Peso total da Árvore Geradora Mínima: 42488.0
Peso total da Solução aproximada: 55583.0
Peso total da Solução ótima: 48912

Fator de Aproximação: 55583.0/48912 ≈ 1.136 vezes o ótimo

4.1.6 eil76

Peso total da Árvore Geradora Mínima: 463.0
Peso total da Solução aproximada: 634.0
Peso total da Solução ótima: 538

Fator de Aproximação: 634.0/538 ≈ 1.178 vezes o ótimo

4.1.7 fl417

Peso total da Árvore Geradora Mínima: 10151.0
Peso total da Solução aproximada: 13635.0
Peso total da Solução ótima: 11861

Fator de Aproximação: 13635.0/11861 ≈ 1.149 vezes o ótimo

4.1.8 lin318

Peso total da Árvore Geradora Mínima: 37906.0
Peso total da Solução aproximada: 48233.0
Peso total da Solução ótima: 42029

Fator de Aproximação: 48233.0/42029 ≈ 1.147 vezes o ótimo

4.1.9 si175

Peso total da Árvore Geradora Mínima: 20762.0
Peso total da Solução aproximada: 21999.0
Peso total da Solução ótima: 21407

Fator de Aproximação: 21999.0/21407 ≈ 1.028 vezes o ótimo

Mesmo peso de árvore mínima fornecido pelo professor

4.1.10 si535

Peso total da Árvore Geradora Mínima: 47552.0
Peso total da Solução aproximada: 50486.0
Peso total da Solução ótima: 48450

Fator de Aproximação: 50486.0/48450 ≈ 1.042 vezes o ótimo

5. Conclusão

Em todas as instâncias testadas, o algoritmo de Christofides produziu soluções aproximadas que ficaram dentro do limite teórico de 1.5 vezes a solução ótima. O resultado das Árvores Geradoras Mínimas das instâncias de testes disponibilizadas pelo professor foi o mesmo encontrado pelo nosso algoritmo, as outras 8 instâncias do repositório foram testadas em implementações confiáveis de christofides de terceiros, e os resultados também se mostraram consistentes.

Concluímos que a nossa implementação do algoritmo de Christofides em produzir rotas de Caixeiro Viajante com custos muito próximos ao ideal para grafos métricos, é uma solução válida e eficaz.