Relatório do Trabalho Prático N.02

Pedro Malta Boscatti[1], **Rodrigo de Oliveira Santos** [2], **Sophia Damasceno Satuf**[3]

[1]Instituto de Ciências Exatas e de Informática
Pontifícia Universidade Católica de Minas Gerais (PUC Minas)

Resumo. *O presente documento é um registro detalhado do desenvolvimento do Trabalho Prático N. 02 que visa executar uma variedade de algoritmos, conforme as exigências da disciplina de Teoria dos Grafos e Computabilidade. Ao longo do registro, serão documentadas escolhas de implementação, soluções desenvolvidas, testes implementados e conclusões obtidas. O objetivo é alcançar uma compreensão abrangente dos conceitos fundamentais e a capacidade de aplicar esse conhecimento em contextos práticos.*

1. Introdução

O problema dos *k*-centros representa uma tarefa clássica da análise de dados sendo sua solução relacionada às técnicas de clustering. Neste problema, dado um grafo completo com custos nas arestas (geralmente, respeitando a desigualdade triangular) e um inteiro positivo *k*, deseja-se encontrar um conjunto de *k* vértices (chamados centros) que minimize a maior distância de um vértice qualquer do grafo ao conjunto de centros (isto é, a maior distância de um vértice até um dos centros selecionados). Essa distância é, por vezes, denominada de raio da solução.

Dessa forma, essa tarefa lida com a questão de se particionar elementos em conjuntos (ou clusters) de acordo com suas (dis)similaridades (representadas pelas distâncias entre vértices). Trata-se, portanto, de um problema bastante genérico que aparece em diversas aplicações (além de existirem muitas variantes dele na literatura). A principal diferença entre essas variantes é a função objetivo (ou melhor, o critério utilizado para se determinar a solução ótima). Uma definição formal do problema dos *k*-centros pode ser dada da seguinte forma:

Problema dos *k*-centros: Dada uma coleção de V pontos (ou vértices) com distâncias definidas para cada par de pontos pela função $d : V \times V \rightarrow \mathbb{R}^+$, o objetivo é encontrar um conjunto $C \subseteq V$, com $|C| \leq k$, cujos elementos são chamados de centros, de forma que a maior distância de um ponto (ou vértice) $v \in V$ ao centro mais próximo seja mínima.

Vale notar que nessa formulação, fixada uma instância (V, d, k), uma escolha de centros $C = \{c_1, c_2, \ldots, c_p\} \subseteq V$ fornece implicitamente uma partição de V em $p \leq k$ conjuntos $C1, C2, \ldots, Cp$ em que $C_i = \{v \in V \mid d(v, c_i) \leq d(v, c_j), \forall j \neq i\}$ para $i = 1, 2, \ldots, p$. Se para um dado ponto (ou vértice) v houver mais de um centro que atinge a distância mínima, escolhe-se um deles arbitrariamente para determinar o cluster de v.

Uma possível interpretação desse problema seria a automatização de uma processo de divisão em categorias. Pode-se imaginar, como exemplo, o uso de dados de compras de consumidores. De acordo com os tipos e quantidades de produtos comprados, poderia ser elaborada uma maneira de se medir a ”distância” entre dois consumidores. Assim,

o problema dos k-centros pode ser visto como o problema de se estabelecer k ”perfis de consumidor” de forma que os consumidores de um dado peril (cluster) formem um grupo o mais homogêneo possível. Essa mesma ideia pode ser explorada para se categorizar alunos de um curso, usuários de uma rede social, deputados de uma casa legislativa, entre outros cenários, utilizando critérios de comparação que modelem mais convenientemente cada situação.

Outra interpretação é como uma tarefa de localização de facilidades. Os pontos (ou vértices) representam um conjunto de locais que precisam ser supridos por certo tipo de instalação e suas distâncias são definidas pela rota de menor custo de um ponto a outro - em que o custo pode ser a distância no espaço, o tempo, ou uma combinação desses e outros fatores. Deseja-se, assim, ”abrir” tais instalações em até k pontos de maneira que nenhum ponto esteja muito longe da instalação mais próxima. Por exemplo, imagine que se tem um conjunto de hospitais públicos e que se dispõe de recursos para abrir centros para tratamento de uma doença (como, um câncer) em no máximo k deles. Deseja-se determinar em quais hospitais devem ser instalados tais centros de modo a minimizar o maior tempo de transporte entre um hospital ”comum” e seu centro de tratamento mais próximo, a fim de que a transferência mais remorada de um paciente seja tão rápida quanto possível.

A resolução do problema dos k-centros de forma exata pode ser difícil (ou mesmo, inviável para grandes instância)[4], contudo ela pode ser feita por diferentes abordagens aproximadas.

2. Desenvolvimento

Nesta seção, detalhamos a implementação de um algoritmo em Python para resolver o problema dos K-centros em grafos. O algoritmo lê grafos a partir de arquivos de entrada, aplica o algoritmo de Floyd-Warshall para determinar as menores distâncias entre todos os pares de vértices, e encontra a combinação de K vértices que minimiza o raio máximo dos clusters. A implementação é descrita em detalhes, com explicações sobre cada bloco de código.

2.1. OR-Library

Antes de detalharmos o desenvolvimento propriamente dito, vamos descrever a coleção utilizada para o posterior teste de algoritmos. A OR-Library[3] é uma coleção de instâncias de problemas de otimização comumente estudados na pesquisa operacional, disponibilizada online para fins acadêmicos e de pesquisa. Ela inclui problemas como roteamento de veículos, problemas de escalonamento, problemas de localização, entre outros. Para a fase de testes das implementações utilizamos as 40 instâncias disponíveis para o problema ”*p-median - uncapacitaded*”[1]. A Tabela 1 apresenta para cada instância seu tamanho (em número de vértices, isto é, $|V|$), o número máximo de centros a serem encontrados (isto é, k) e o valor de raio da solução ótima.

[1]Os arquivos com as 40 instâncias são denominados ”pmed1.txt”, ”pmed2.txt”, ..., ”pmed40.txt”.

Table 1. Lista de instâncias com seus tamanhos e valores de raio.

Instância	$\|V\|$	k	Raio
01	100	5	127
02	100	10	98
03	100	10	93
04	100	20	74
05	100	33	48
06	200	5	84
07	200	10	64
08	200	20	55
09	200	40	37
10	200	67	20
11	300	5	59
12	300	10	51
13	300	30	35
14	300	60	26
15	300	100	18
16	400	5	47
17	400	10	39
18	400	40	28
19	400	80	18
20	400	133	13
21	500	5	40
22	500	10	38
23	500	50	22
24	500	100	15
25	500	167	11
26	600	5	38
27	600	10	32
28	600	60	18
29	600	120	13
30	600	200	9
31	700	5	30
32	700	10	29
33	700	70	15
34	700	140	11
35	800	5	30
36	800	10	27
37	800	80	15
38	900	5	29
39	900	10	23
40	900	90	13

2.2. Leitura e Inicialização do Grafo

Primeiramente, lemos os arquivos de entrada contendo a descrição dos grafos. O nome da pasta contendo os arquivos deve ser guardado na variável ”pasta_contendo_as_entradas”.

Deve-se colocar barra dupla para que o código funcione. Para o arquivo de resposta, deve ser fornecido o nome da pasta e o nome do arquivo nas variáveis, ”pasta_contendo_as_respostas” e ”nome_arquivo_resposta” respectivamente.

```

pasta_contendo_as_entradas = r"D:\\Faculdade\\Implementa o Grafos\\Problema-dos-K-Centros-\\docs\\entradas"
pasta_contendo_as_respostas = r"D:\\Faculdade\\Implementa o Grafos\\Problema-dos-K-Centros-\\docs\\resposta"

nome_arquivo_resposta = "resposta.txt"
for nome_do_arquivo in os.listdir(pasta_contendo_as_entradas):


    if nome_do_arquivo.endswith('.txt'):



        conta_arquivos = 1


        caminho_do_arquivo_resposta = os.path.join(pasta_contendo_as_respostas,nome_arquivo_resposta)

        with open(caminho_do_arquivo_resposta,'r') as arquivo_resposta:

            instrucoes_resposta = arquivo_resposta.readlines()

            resposta_da_instancia = instrucoes_resposta[conta_arquivos-1].split()

            raio_solucao = int(resposta_da_instancia[-1])


        caminho_do_arquivo = os.path.join(pasta_contendo_as_entradas, nome_do_arquivo)
        nome_do_grafo = f'grafo{caminho_do_arquivo}'
```

Cada arquivo contém informações sobre o número de vértices, arestas e clusters. Utilizamos uma matriz de adjacência para representar o grafo, onde inicialmente todas as distâncias são definidas como infinito, exceto a distância de um vértice para ele mesmo, que seria zero.

```
import os
import numpy as np
from itertools import combinations

INFINITO = np.inf

for nome_do_arquivo in os.listdir(pasta_contendo_as_entradas):
    if nome_do_arquivo.endswith('.txt'):

        nome_do_grafo = f'grafo{caminho_do_arquivo}'

        with open(caminho_do_arquivo,'r') as arquivo:
            instrucoes = arquivo.readline().split()
```

```
            n_vertices = int(instrucoes[0])+1
            n_arestas = int(instrucoes[1])+1
            n_clusters = int(instrucoes[2])

            grafo = np.full((n_vertices,n_vertices),INFINITO)
```

Em seguida, o restante das linhas é processado para que o vértice de origem e o vértice de destino e a distância entre eles seja armazenada na matriz de adjacência *"grafo"*. É também nesse trecho que a distância do vértice para ele mesmo é inicializada como zero e a distância entre vértices com arestas inexistentes como infinita, como citado anteriormente.

```
with open(caminho_do_arquivo,'r') as arquivo:

    for linha in arquivo:
        origem_destino_distancia = linha.split()
        origem = int(origem_destino_distancia[0])
        destino = int(origem_destino_distancia[1])
        distancia = int(origem_destino_distancia[2])
        grafo[origem][destino] = distancia
        grafo[origem][origem] = 0
```

Com a construção em mãos, já é possível seguir para a implementação e teste dos algoritmos que melhor atenderão a proposta do trabalho.

2.3. Aplicação do Algoritmo de Floyd-Warshall

Após a construção do grafo, via matriz, aplicamos o algoritmo *Floyd-Warshall*[2]. Com ele, é possível obter a distância para todos os pares de vértices[1].

Para a execução do mesmo, inicializamos antes uma estrutura contendo as distâncias entre os vértices, recebida da matriz *grafo* e iterando sobre ela para guardar os vértices de destino para os determinados vértices de origem. Com isso, os melhores caminhos serão guardados iteração a iteração.

```
matriz_distancia_iteracao_anterior = grafo

    matriz_com_caminhos_iteracao_anterior = [[INFINITO]*n_vertices for
    vertice in range(n_vertices)]
```

Na sequência, o algoritmo *Floyd-Warshall* é executado de fato e, com isso, descobrimos se existe um caminho entre um vértice e outro. Caso não exista, a distância assume um valor infinito (no caso da implementação, um número inteiro muito alto) e, caso exista, a posição de destino será preenchida com o vértice de origem.

```
for origem in range(n_vertices):
    for destino in range(n_vertices):
        if(0 < matriz_distancia_iteracao_anterior[origem][destino] <
    INFINITO):
            matriz_com_caminhos_iteracao_anterior[origem][destino] =
    origem
```

Finalizado o método acima, inicializamos as matrizes para guardar os valores das iterações. Desta maneira, se a distância da iteração anterior for maior, o valor na matriz

final deverá ser atualizado. Alguns comentários com mais informações intermediárias também podem ser vistas no trecho do código.

```
for linha in range (n_vertices):
    for coluna in range(n_vertices):
        for iteracao in range(n_vertices):
            if matriz_distancia_iteracao_anterior[coluna][iteracao] <= matriz_distancia_iteracao_anterior[coluna][linha] + matriz_distancia_iteracao_anterior[linha][iteracao]:
                matriz_distancias_iteracao_atual[coluna][iteracao] = matriz_distancia_iteracao_anterior[coluna][iteracao]
                matriz_caminhos_iteracao_atual[coluna][iteracao] = matriz_caminhos_iteracao_atual[coluna][iteracao]
            else:
                matriz_distancias_iteracao_atual[coluna][iteracao] = matriz_distancias_iteracao_atual[coluna][iteracao] + matriz_distancias_iteracao_atual[iteracao][coluna]
                matriz_caminhos_iteracao_atual[coluna][iteracao] = matriz_com_caminhos_iteracao_anterior[linha][iteracao]
            ## Apos realizar os calculos dos novos pesos e tambem determinar os melhores caminhos para a iteracao atual, as matrizes devem ser atualizadas
            for linha in range(n_vertices):
                for coluna in range(n_vertices):
                    matriz_distancia_iteracao_anterior[linha][coluna] = matriz_distancias_iteracao_atual[linha][coluna]
                    matriz_com_caminhos_iteracao_anterior[linha][coluna] = matriz_caminhos_iteracao_atual[linha][coluna]
```

Para saber a distância do vértice *i* até o vértice *j*, ou seja, o seu custo, deve-se utilizar a matriz *matriz_caminhos_iteracao_atual*. Ao final do programa, essa matriz terá a distância de todos os vértices.

Como exemplo, a distância para ir do vértice 2 até o vértice 6 poderia ser obtida da seguinte forma:

distância = matriz_caminhos_iteracao_atual[2][6]

2.4. Cálculo das Combinações de K Centros

A próxima etapa do código consiste em realizar todas as combinações possíveis de *n* clusters em *t* números de vértices. Desta maneira, para um conjunto de 100 vértices, por exemplo, todas as combinações possíveis para 5 clusters resulta em 75.287.520 conjuntos possíveis.

A primeira combinação irá conter os centróides possíveis. O algoritmo itera sobre os vértices e sobre os centros comparando-se as distâncias entre os pares vértice-centro. Aquela que for a menor distância será o cluster desse vértice. Essa distância é utilizada para uma possível atualização da lista dos *raios_clusters*, inicializada com zeros. Ao terminar a primeira iteração para o primeiro vértice esse valor raio do cluster é atualizado. Para o próximo vértice essa distância é comparada com a anterior. Caso a distância atual seja maior que a anterior, a lista *raios_clusters* é atualizada.

Ao percorrer todos os vértices, terei apenas o raio máximo armazenado na lista *raios_clusters*. O raio da combinação é maior raio contido na lista *raios_clusters*.

No início do código abaixo é definido a *melhor_combinacão* e o *raio_resposta* como zeros. Esses valores são utilizados ao final do processamento para comparar a iteração atual com a anterior. A ideia é sempre comparar os raio associados a cada combinação. Aquela combinação que possui o raio mais próximo da resposta será o valor da variável *combinação_resposta* e o seu raio a variável *raio_resposta*.

Assim, ao final do processamento terei apenas a lista contendo os vértices centróides que retornaram a melhor combinação.

```
    numeros = list(range(n_vertices))
    melhor_combinacao = [0,0,0,0]
    raio_resposta = 0
    for combinacao in combinations(numeros,n_clusters):

        #transformando a tupla em uma lista
        centroides_selecionados = [elemento for elemento in combinacao]
        # Criando uma lista para conter os elemento do cluster
        clusters = [elemento for elemento in centroides_selecionados]

        raio_clusters = [0 for elemento in centroides_selecionados]

        for vertice in n_vertices:
            # Por exemplo, para o v rtice 1, devo saber qual a menor
    dist ncia entre os centr ides selecionados
            # Assim tenho que verificar cada dist ncia em rela  o ao
     centr ide e pegar a m nima
            distancia_minima = INFINITO

            for centroide in centroides_selecionados:
                # Aqui busco a dist ncia do v rtice atual para o
    centr ide selecionado
                distancia_temporaria = matriz_caminhos_iteracao_atual[
    centroide][vertice]
                # Se a distancia da itera  o for menor do que a
    distancia minima
                # A distancia minima    atualizada em conjunto com o
    centroide
                if distancia_temporaria <= distancia_minima:
                    distancia_minima = distancia_temporaria
                    centroide_do_vertice = centroide
            # Ap s a itera  o irei saber em qual cluster esse
    v rtice pertence
            posicao_do_centroide_na_lista = clusters.index(centroide)
            # Ap s encontrar a posi  o do centr ide
            # Devo adicionar a distancia m xima, ou seja, o raio para
    aquele cluster na lista dos raios
            raio_temporario = distancia_minima
            if raio_clusters[posicao_do_centroide_na_lista] <=
    raio_temporario:
                raio_clusters[posicao_do_centroide_na_lista] =
    raio_temporario

        #Ent o ap s todo o processamento queremos saber apenas qual o
     maior raio dentre todos os clusters
        # Ou seja o maior valor dentro da lista raio_clusters.
        #ao final o raio da solu  o    o maior raio entre esses
```

```
    clusters
        maior_raio = max(raio_clusters)

        #Devo guardar a combina  o para no final conferir qual foi a
    combina  o que gerou o resultado mais pr ximo do
        #esperado
        #Aqui estou comparando a resposta do raio anterior com o raio
    atual
        alvo = raio_solucao
        combinacao_temporaria = melhor_combinacao
        raio_anterior = raio_resposta
        raio_atual = maior_raio
        if abs(alvo - raio_anterior) < abs(alvo - raio_atual):
            combinacao_resposta = combinacao_temporaria
            raio_resposta = raio_anterior
        else:
            combinacao_resposta = combinacao
            raio_resposta = raio_atual


        print(f'Arquivo: {nome_do_grafo}')
        print(f'melhor combinacao: {combinacao_resposta}')
        print(f'raio da combinacao: {raio_resposta}')
```

3. Solução Aproximada

Nesta seção, detalhamos a implementação de um algoritmo em Python para resolver de forma aproximada o problema dos K-centros em grafos. O método recebe os grafos da biblioteca OR-Library[3] e aplica o algoritmo k-means repetidamente para aproximar o conjunto de K vértices que minimiza o raio máximo dos clusters. A implementação é descrita em detalhes, com explicações sobre cada bloco de código.

3.1. Algoritmo K-means

O algoritmo de K-means pode ser utilizado para formular uma solução aproximada para o problema dos k-centros uma vez que seu algoritmo tenta minimizar a soma das distâncias quadradas (variância) dos pontos aos seus centros mais próximos. Enquanto o problema dos k-centros visa minimizar o raio máximo dos clusters, o algoritmo k-means tende a encontrar clusters mais compactos em termos de soma de distâncias ao centro, mas não necessariamente minimiza a maior distância ao centro. Com isso em mente, uma forma de utilizar o k-means para aproximar a solução do problema dos k-centros é ajustando a inicialização e/ou executando o algoritmo múltiplas vezes em busca de uma solução melhor.

Primeiramente, o algoritmo recebe o grafo referente à instância que está sendo testada assim como o seu número k. Então, ele escolhe arbitrariamente k centroides iniciais dos nós do grafo.

```
def kmeans(graph, k):
    num_nodes = len(graph)

    # incializa centroids
    centroids = random.sample(range(num_nodes), k)
```

Logo após, o algoritmo faz a atribuição inicial dos clusters, atribuindo cada nó do grafo ao cluster cujo centróide é o mais próximo. Essa atribuição é baseada nas distâncias entre os nós, que são pré-calculadas e armazenadas na matriz graph.

O método assign_clusters inicialmente cria um dicionário chamado clusters onde cada chave é um centróide e os valores são listas vazias que representarão os nós pertencentes a esses clusters. Para cada nó no grafo, o método calcula a distância para cada centróide, comparando-as para encontrar a menor distância. O nó é então adicionado à lista correspondente ao centróide mais próximo. Dessa forma, ao final da execução do método, cada nó do grafo estará associado ao cluster do centróide mais próximo, proporcionando uma partição do grafo baseada na proximidade a esses centróides.

```
    # define os clusters para os centroides
    clusters = assign_clusters(graph, k, centroids)
```

```
def assign_clusters(graph, k, centroids):
    num_nodes = len(graph)
    clusters = {centroid: [] for centroid in centroids}

    for i in range(num_nodes):
        min_dist = INF
        clossest_centroid = None

        # passa por todos os nos do grafo e coloca-os no
        # cluster de centroide mais proximo
        for j in range(k):
            if graph[i][centroids[j]] < min_dist:
                min_dist = graph[i][centroids[j]]
                clossest_centroid = centroids[j]
        clusters[clossest_centroid].append(i)

    return clusters
```

Então é chamado o método find_radius, que calcula a maior distância entre o centróide do cluster e qualquer outro nó dentro desse cluster. Inicialmente, o raio é definido como zero, pois a função procura o valor máximo. Para cada cluster no dicionário, a função inicializa um max_radius também como zero. Em seguida, para cada nó, a função verifica se a distância entre o nó e a chave do cluster é maior que o max_radius atual. Se sim, ela atualiza o max_radius.

Depois de verificar todos os nós em todos os clusters, a função compara o max_radius de cada cluster com o definitive_radius atual e atualiza o definitive_radius se o max_radius for maior. Por fim, a função retorna o definitive_radius, que representa o maior raio encontrado entre todos os clusters. Após a primeira chamada desse método, os clusters iniciais são atribuídos à variável best_clusters.

```
    # encontra o raio dos centroides
    radius = find_radius(graph, clusters)

    # atribui o melhor cluseter aos clusters inciciais
    best_clusters = clusters
```

```
    def find_radius(graph, clusters):
    # definir o raio inicial como zero porque queremos o maximo
```

```
    definitive_radius = 0
    for key, cluster in clusters.items():
        # o raio maximo sera calculado para o proprio cluster
        max_radius = 0
        for node in cluster:
            # se a distancia for maior que o raio anterior
            # definir o novo raio
            max_radius = max(graph[key][node], max_radius)
        # comparar o raio do cluster com o raio da solucao
        definitive_radius = max(max_radius, definitive_radius)

    return definitive_radius
```

Para cada cluster, o algoritmo recalcula o centróide. O novo centróide é determinado como o nó do cluster que minimiza a distância média para todos os outros nós dentro do mesmo cluster. Essa atualização contínua dos centroides garante que cada centróide se mova em direção ao centro do cluster, resultando em uma melhor representação do centro dos dados agrupados. Com os novos centroides determinados, o algoritmo procede para reatribuir todos os nós aos clusters de acordo com o centróide mais próximo. Esta etapa é crucial para garantir que cada nó esteja no cluster mais adequado, baseado na proximidade ao centróide, permitindo uma melhor estruturação dos dados dentro dos clusters.

O raio da nova solução é recalculado com base nos novos centroides e na nova atribuição dos clusters. Esta medida é usada para avaliar a compactação do cluster e a eficácia da atribuição dos nós aos centroides. Um raio menor indica clusters mais compactos e bem definidos.

```
    while True:
        # trocar o centroide pelo no mais proximo do centro do cluster
        new_centroids = update_centroids(graph, clusters)

        centroids = new_centroids
        # atribuir o cluster para os novos centroides
        clusters = assign_clusters(graph, k, centroids)
        # definir o novo raio tambem
        new_radius = find_radius(graph, clusters)

        # parar o kmeans para nao ficar em loop infinito
        if set(new_centroids) == set(centroids):
            # se a solucao nao mudar, sair do loop
            break
        else:
            # se a solucao for melhor, definir como a melhor solucao
            best_clusters = clusters
            radius = new_radius

    # retornar a melhor solucao encontrada
    return best_clusters, radius
```

O algoritmo continua a atualizar os centroides e a reatribuir os clusters até que os centroides não mudem mais. Esse critério de parada é fundamental para evitar loops infinitos e garantir a convergência do algoritmo. Quando os centroides não mudam mais entre iterações, significa que a melhor estrutura de clusters foi alcançada, e o algoritmo pode parar com confiança. Finalmente, o algoritmo retorna os clusters finais e o raio da

melhor solução encontrada.

```
def update_centroids(graph, clusters):
    new_centroids = []
    for cluster in clusters.values():
        min_avg_dist = INF
        new_centroid = None

        for node in cluster:
            # define a media do no para todos os outros nos no cluster
            avg_dist = np.mean([graph[node][other] for other in cluster
    ])

            # se tiver a menor media de distancia, definir como ele
            if avg_dist < min_avg_dist:
                min_avg_dist = avg_dist
                new_centroid = node
        # para cada cluster, escolher o no com a menor media de
    distancia para os outros
        # nos como o novo centroide
        new_centroids.append(new_centroid)

    return new_centroids
```

4. Testes e Resultados

Assim como exigido nas orientações do trabalho, foram feitas testagens sobre os dois métodos desenvolvidos e utilizados. Dessa forma, foi possível ver o desempenho de ambos com o crescente número de possibilidades de grafos fornecidos nos arquivos *.txt*.

Para a verificação foram testados os quarenta grafos colocados dentro da pasta *entradas* em *docs*, na estrutura do arquivo.

4.1. Processo de Testagem

O processo de testagem nesse trabalho consistiu em organizar os arquivos da OR-Library em uma pasta para se ajustar ao código e observar como o programa se comportava e imprimia os tempos no arquivo de texto de log, para que pudéssemos comparar.

4.2. Resultados Obtidos

Os resultados obtidos e exibidos na tabela abaixo, foram conseguidos por meio do algoritmo K-Means, de raio aproximado.

Os resultados de força bruta não puderam ser exibidos aqui, pois o código já demonstrado na seção de **Desenvolvimento** não conseguiu gerar resultado em tempo hábil, estourando a capacidade de memória dos computadores ou demorando um tempo muito longo para finalizar a iteração. Isso demonstra a necessidade da continuidade da pesquisa e ajustes de otimização futuros para obter um resultado mais eficaz.

Table 2. Tempos de execução e solução aproximada para cada instância.

Tempo [s]	Instância	$\|V\|$	k	Raio Aproximado	Raio
0.0019993782043457030	01	100	5	197	127
0.0019984245300292970	02	100	10	145	98
0.0010015964508056640	03	100	10	196	93
0.0009996891021728516	04	100	20	133	74
0.0020031929016113280	05	100	33	141	48
0.0051081180572509766	06	200	5	113	84
0.0035660266876220703	07	200	10	104	64
0.0030004978179931640	08	200	20	102	55
0.0050082206726074220	09	200	40	74	37
0.0069997310638427734	10	200	67	57	20
0.0105025768280029300	11	300	5	85	59
0.0064370632171630860	12	300	10	85	51
0.0060002803802490234	13	300	30	68	35
0.0090003013610839840	14	300	60	76	26
0.0119996070861816400	15	300	100	44	18
0.0156421661376953120	16	400	5	65	47
0.0095131397247314450	17	400	10	68	39
0.0100927352905273440	18	400	40	60	28
0.0140035152435302730	19	400	80	46	18
0.0210053920745849600	20	400	133	47	13
0.0139999389648437500	21	500	5	61	40
0.0115754604339599610	22	500	10	71	38
0.0130004882812500000	23	500	50	44	22
0.0210013389587402340	24	500	100	40	15
0.0354399681091308600	25	500	167	48	11
0.0215666294097900400	26	600	5	55	38
0.0187335014343261720	27	600	10	50	32
0.0190041065216064450	28	600	60	71	18
0.0285065174102783200	29	600	120	38	13
0.0430026054382324200	30	600	200	40	9
0.0299515724182128900	31	700	5	48	30
0.0190064907073974600	32	700	10	86	29
0.0230000019073486330	33	700	70	31	15
0.0385909080505371100	34	700	140	45	11
0.0435123443603515600	35	800	5	45	30
0.0335054397583007800	36	800	10	67	27
0.0309965610504150400	37	800	80	36	15
0.0689122676849365200	38	900	5	57	29
0.0387523174285888700	39	900	10	31	23
0.0365095138549804700	40	900	90	27	13

5. Conclusão

Neste estudo, testou-se a eficiência de dois algoritmos distintos para encontrar uma resolução para o problema dos *k-centros*. Foi observado que, entre os métodos avaliados - resolução exata e resolução aproximada - este último se destacou como o mais eficiente. Sua abordagem demonstrou uma capacidade notável de lidar com grafos de diferentes tamanhos e complexidades, oferecendo resultados consistentes e tempos de execução significativamente mais rápidos.

O tempo superior apresentado pelo algoritmo é atribuído ao fato de que ele faz menos iterações ao longo dos grafos, possibilitando um consumo de memória e tempo muito menor. Porém, como o próprio nome já diz, seus resultados são *aproximados*. Esses achados ressaltam a importância de escolher estratégias algorítmicas apropriadas para resolver problemas específicos em grafos.

Ao encerrar este registro do desenvolvimento do trabalho centrado na execução de diversos algoritmos, emerge uma apreciação profunda da complexidade e da diversidade presente na Ciência da Computação. Através da implementação prática dos algoritmos exigidos, consolidamos não apenas o conhecimento teórico adquirido durante a disciplina, mas também cultivamos habilidades valiosas em análise, resolução de problemas e programação.

As escolhas de implementação, os desafios superados e as soluções criadas destacam não apenas a aplicação prática dos conceitos estudados, mas também a necessidade de flexibilidade e adaptação diante de problemas do mundo real. A base predeterminada serviu como alicerce, oferecendo um ponto de referência constante e enfatizando a importância de construir sobre fundamentos sólidos.

References

[1] Floyd-warshall algorithm. `https://cp-algorithms.com/graph/all-pair-shortest-path-floyd-warshall.html`. Acesso em junho de 2024.

[2] Implementação em python do algoritmo de floyd-warshall. `https://www.youtube.com/watch?v=FzBs8BV2oHs`. Acesso em junho de 2024.

[3] Or-library. `http://people.brunel.ac.uk/~mastjjb/jeb/orlib.html`. Acesso em junho de 2024.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 3rd edition, 2009.