PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

Trabalho Prático 1*

Model - Pratical Exercise - ICEI - Puc Minas

Davi Dias Magalhães[1]
Henrique Castro[2]
Leonardo Gomide[3]

Resumo

Este exercício prático contem a documentação do Trabalho Prático I, feita em LaTeX. O problema dos K-centros representa uma tarefa clássica da análise de dados sendo sua solução relacionada às técnicas de clustering. Essa tarefa lida com a questão de se particionar elementos em conjuntos de acordo com suas (dis)similaridades. A principal diferença entre essas variantes é o critério utilizado para se determinar a solução ótima.
Neste trabalho foram implementadas e comparadas duas formas distintas para resolução do problema dos k-centros. Uma delas garantidamente encontra a melhor resposta, porem não é eficiente. E a outra encontra rapidamente uma aproximação da resposta entretanto não necessariamente será a resposta exata. As implementações foram testadas com as instâncias disponíveis para OR-Library.

Palavras-chave: LaTeX. Exercício Prático. Java.

*Artigo apresentado como documentação ao Instituto de Ciências Exatas e Informática da Pontifícia Universidade Católica de Minas Gerais para disciplina Teoria dos Grafos e Computabilidade.

[1]Curso de Ciência da Computação PUC Minas, Brasil– https://github.com/davidias3
[2]Curso de Ciência da Computação PUC Minas, Brasil– https://github.com/HenriqueCastros
[3]Curso de Ciência da Computação PUC Minas, Brasil– https://github.com/GomideLeo

1 INTRODUÇÃO

Tarefas de agrupamentos são comuns e diversamente aplicadas em diferentes contextos, sejam esses computacionais ou de outros âmbitos. Dentre elas existe a o problema dos K-Centros, que consiste em encontrar um conjunto K de vértices, ou centros, que minimize a maior distância de um vértice qualquer ao conjunto de escolhido. Essa distância pode ser referida como raio.

Neste trabalho estaremos explorando possíveis soluções para esse problema. No intuito de testar e avaliar o desempenho dos nossos algoritmos utilizaremos as instâncias disponíveis pela **OR-Library**. Ao fim do trabalho, vamos comparar nossos resultados com os resultados corretos de cada instância, podendo ver o quão correto estariam nossas aproximações.

2 IMPLEMENTAÇÃO

Para desenvolver a solução do trabalho, utilizamos a linguagem Java. Ela foi escolhida por ser uma linguagem que permite implementar facilmente orientação de objetos e ser relativamente rápida.

Foram criadas diferentes classes que se complementam na solução, das quais se destacam as classes *Grafos* e as classes que implementam os algoritmos, *BruteForceSolver* e *MST-Solver*. Todas as classes foram escritas de forma a rodar em uma única *Thread* de execução.

2.1 Classe Grafos

A classe Grafo foi desenvolvida para ser a principal estrutura de dados utilizada durante o trabalho. Apesar dos exercícios serem para grafos não direcionados, implementamos a classe de forma que ela também para grafos direcionados, o atributo *isDirectional* faz o controle dessa característica do grafo, tendo o *default* como falso.

Utilizamos de uma matriz de adjacência para armazenar as arestas e seus respectivos pesos, dentro da classe a matriz pode ser acessada por meio do atributo *edgesWeights*. Fora a facilidade de utilização e implementação, a matriz de adjacência permite calcular os caminhos mínimos por meio do algoritmo de Floyd-Warshall de forma simples, a matriz resultante desse algoritmo pode ser gerada por meio do método *getMinDistanceMatrix*.

Outros métodos que valem destacar são *calculateExentricity* e *isReachableFrom*. O primeiro método calcula a excentricidade de um dado nó, que é usado em alguns métodos de outras classes que serão apresentadas. Para chegar ao resultado desse método utilizamos o algoritmo de busca de custo uniforme, que garante encontrar uma resposta se possível. O segundo método retorna uma valor booleano, que indica se dada uma origem é possível atingir outro vértice, por meio da busca em largura.

Fora os métodos explicitados acima, construímos outros métodos auxiliares que utilizam pesos na matriz de adjacência, adicionar vértices ou arestas, e outras funções que julgamos necessárias.

2.2 Algoritmos

Para resolver o problema, a primeira abordagem a se tomar é a força bruta, que garantidamente encontrará a melhor solução ao custo da performance do algoritmo. Ela consiste de testar todas as combinações possíveis dos K-Centros e escolher a combinação que possui o melhor resultado. A classe responsável por realizar essa solução é a *BruteForceSolver*.

A segunda abordagem que encontra uma solução é buscar uma aproximação, que não garantidamente vai encontrar os melhor centros para um problema mas performa de forma mais rápida que a outra solução. A classe que implementa essa forma de solução é a *MSTSolver*.

2.2.1 Algoritmo de Força Bruta

A classe *BruteForceSolver* é a classe responsável por encontrar, garantidamente, uma resposta. Para tal, implementamos um algoritmo de força, não otimizado, e por essa razão em alguns cenários de teste, nossa solução saiu por *TIMEOUT*, como será abordado na seção 3.

Por ser um algoritmo de força bruta, o seu funcionamento é relativamente simples. Para cada k calculamos quais os melhores centros e uma vez tendo calculado para todos os k's, calculamos com qual quantidade de centros encontramos o melhor resultado.

O método responsável por encontrar os melhores para um dado k é *findBestCenterForN*, e sua variante iterativa *findBestCenterForNIterative*. Para cada combinação de centros possível, o método distribui os vértices para os centros mais próximos, por meio do método *distributeNodesToCenters*, e calcula o maior raio dessa distribuição, *findMaxRadiusOfDistribution*. Uma variável auxiliar é utilizada para guardar qual o menor raio encontrado até então e quais os centros da combinação. Em primeiro momento geramos as combinações de forma recursiva e guardávamos em uma variável. Contudo, assim que foi testado em um grafo, percebeu-se que o método estourava a memória primária. Por causa disso o método *findBestCenterForNIterative* foi desenvolvido, que gera as combinações de forma iterativa e a cada iteração ele guarda apenas os centros que possuem o menor raio.

2.2.2 Algoritmo de Árvores Geradoras

Para o algoritmo heurístico decidimos utilizar o conceito de árvores geradoras mínimas para obter os componentes de raio mínimo. Ao construirmos uma floresta com as *n-k* meno-

res arestas, podemos garantir um limite superior para o algoritmo, e otimizar essa geração de componentes para obter melhores valores.

Para esse algoritmo inicial foi utilizado como base o algoritmo de Kruskal, onde a quantidade de inserções foi alterada de *|E(T)| - 1* para *|E(T)| - k*, sendo *k* a quantidade de centros desejados, garantindo que *k* árvores serão geradas.

2.2.2.1 Versão 1

Para a primeira versão do algoritmo de geração de floresta, inicializa-se um grafo com todos os vértices do grafo original e uma fila de prioridade com cada uma das arestas do grafo e seus pesos como valor. E a partir dessa fila, as arestas são retiradas em ordem até que *|E(T)| - k* arestas sejam inseridas ou a fila esteja vazia.

Para a inserção, primeiro é verificado se uma das extremidades da aresta é alcançável a partir da outra, descartando a aresta quando for, e inserida temporariamente no grafo quando não for. Se a aresta for inserida, então calcula-se a excentricidade das extremidades da aresta dentro do componente onde se encontram, e é obtida a excentricidade da aresta a partir do mínimo entre esses valores. Essa aresta então é removida da árvore caso sua excentricidade seja maior que o valor da próxima aresta da fila, e retornada com um novo valor igual a sua excentricidade. Isso garante que somente arestas que aumentem pouco a excentricidade dos componentes sejam inseridas.

2.2.2.2 Versão 2

Para a segunda versão, a maior alteração é que o cálculo de excentricidade da aresta ao invés de ser feito somente nas extremidades da aresta, todos os vértices alcançáveis por essas extremidades são buscados e o raio do componente é calculado.

3 ANÁLISE DE RESULTADOS

Nessa seção analisaremos o desempenho de cada um dos algoritmos em relação a tempo e precisão para obter o resultado.

3.1 Experimentos

Para cada instância, executamos cada um dos algoritmos descrito na seção 2.2, em um processador Intel Core i5 7a geração 2.5GHz. Dentre as instâncias a serem testadas, algumas possuem um tamanho considerável, que afetam principalmente o desempenho do algoritmo de

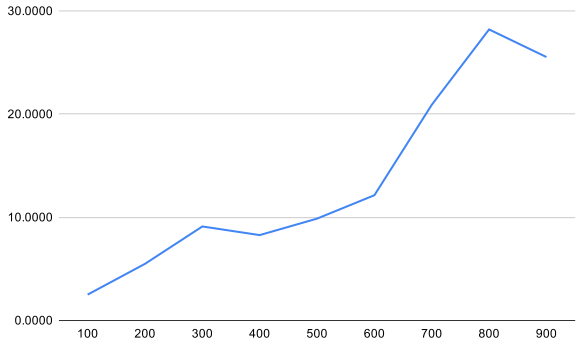

Figura 1 – |V| x % do tempo gasto por V2

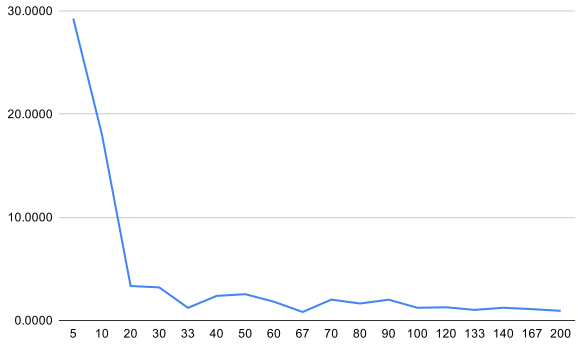

Figura 2 – k x % do tempo gasto por V2

força bruta. Dessa forma para conseguir calcular todos em tempo hábil, determinamos um tempo limite de 1 hora, que se alcançado, a execução é abortada e o melhor resultado até então é retornado.

Na Tabela 1 conseguimos ver os resultados dos algoritmos para cada instância, onde nos casos onde o tempo de execução excedeu o tempo limite temos o rótulo *TIMEOUT*.

3.2 Análise Comparativa

Após o processamento de todos os grafos pelo algoritmo de força bruta, nota-se que para todas as instâncias exceto a 1, obtivemos *TIMEOUT*, isso se dá por conta da quantidade de combinações que o algoritmo de força bruta precisa de calcular para obter o resultado. Para comparação, a instância 1 possui 79.375.495 possíveis combinações de centros enquanto a próxima menor instância (6), possui 2.601.668.490, pouco mais de 32x mais combinações, tornando seu cálculo inviável em tempo hábil.

Tratando dos algoritmos heurísticos, todas as instâncias foram processadas dentro do tempo estipulado tanto para a versão 1 quanto para a 2 do algoritmo de árvores geradoras. Mas nota-se que a versão 2, pela quantidade de operações que precisam ser feitas para calcular o raio dos componentes toma mais tempo para finalizar. Enquanto a versão 1 levou 1 minuto e 26 segundos para processar todas as instâncias, a versão 2 levou 41 minutos e 55 segundos (por volta de 30 vezes mais lento). Essa diferença é acentuada para k pequenos e |V| grandes como pode ser visto nas Figuras 1 e 2

Na questão dos resultados, os dois algoritmos obtiveram resultados com baixa variação, com a versão 1 obtendo um resultado médio de 127% o valor real do raio, e a versão 2 123%, esses resultados também foram regulares se comparados a |V| e k, portanto não há uma perda de qualidade quando trabalhando com instâncias maiores. A diferença entre os resultados dos algoritmos não foi grande, com as maiores melhoras sendo de por volta de 20%. Observando os resultados das instâncias 6, 11, 32 e 38, nota-se que a versão 2 não é sempre melhor que a 1.

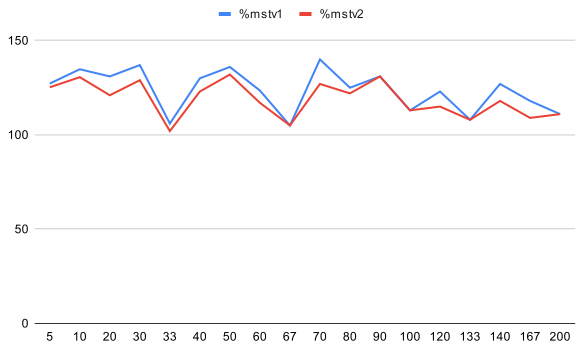

Figura 3 – k x resultados obtidos

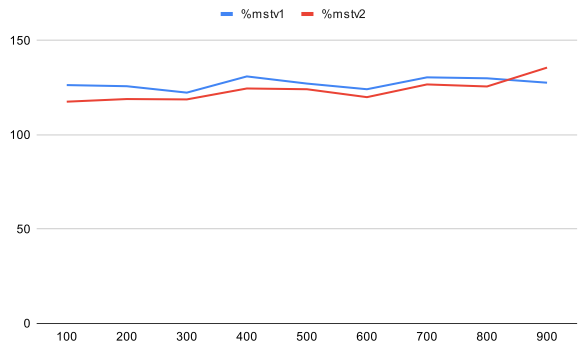

Figura 4 – |V| x resultados obtidos

4 CONCLUSÃO

Tendo em vista a análise acima, diferentes observações podem ser feitas. Apesar de encontrar soluções que possuem uma margem de erro relativamente alta, a solução aproximada conseguiu encontrar solução para todas as soluções sem tomar *TIMEOUT*, em ambas as suas versões.

A solução de força bruta, devido à complexidade das instâncias, foi inefetiva para encontrar soluções. Como observado em seções anteriores, a implementação dos algoritmos foi feita para serem rodados em um única Thread de execução. Uma possível melhoria seria implementar os algoritmos de forma paralela, permitindo a redução do tempo de execução.

Instância	|V|	k	Raio	Força Bruta	MST V1	MST V2
1	100	5	127	127	187 (147%)	156 (123%)
2	100	10	98	TIMEOUT	119 (121%)	118 (120%)
3	100	10	93	TIMEOUT	123 (132%)	116 (125%)
4	100	20	74	TIMEOUT	93 (126%)	87 (118%)
5	100	33	48	TIMEOUT	51 (106%)	49 (102%)
6	200	5	84	TIMEOUT	103 (123%)	111 (132%)
7	200	10	64	TIMEOUT	90 (141%)	77 (120%)
8	200	20	55	TIMEOUT	75 (136%)	68 (124%)
9	200	40	37	TIMEOUT	46 (124%)	42 (114%)
10	200	67	20	TIMEOUT	21 (105%)	21 (105%)
11	300	5	59	TIMEOUT	68 (115%)	71 (120%)
12	300	10	51	TIMEOUT	69 (135%)	65 (127%)
13	300	30	35	TIMEOUT	48 (137%)	45 (129%)
14	300	60	26	TIMEOUT	31 (119%)	29 (112%)
15	300	100	18	TIMEOUT	19 (106%)	19 (106%)
16	400	5	47	TIMEOUT	68 (145%)	60 (128%)
17	400	10	39	TIMEOUT	58 (149%)	54 (138%)
18	400	40	28	TIMEOUT	38 (136%)	37 (132%)
19	400	80	18	TIMEOUT	21 (117%)	21 (117%)
20	400	133	13	TIMEOUT	14 (108%)	14 (108%)
21	500	5	40	TIMEOUT	52 (130%)	51 (128%)
22	500	10	38	TIMEOUT	50 (132%)	50 (132%)
23	500	50	22	TIMEOUT	30 (136%)	29 (132%)
24	500	100	15	TIMEOUT	18 (120%)	18 (120%)
25	500	167	11	TIMEOUT	13 (118%)	12 (109%)
26	600	5	38	TIMEOUT	46 (121%)	46 (121%)
27	600	10	32	TIMEOUT	44 (138%)	42 (131%)
28	600	60	18	TIMEOUT	23 (128%)	22 (122%)
29	600	120	13	TIMEOUT	16 (123%)	15 (115%)
30	600	200	9	TIMEOUT	10 (111%)	10 (111%)
31	700	5	30	TIMEOUT	38 (127%)	35 (117%)
32	700	10	29	TIMEOUT	37 (128%)	42 (145%)
33	700	70	15	TIMEOUT	21 (140%)	19 (127%)
34	700	140	11	TIMEOUT	14 (127%)	13 (118%)
35	800	5	30	TIMEOUT	36 (120%)	35 (117%)
36	800	10	27	TIMEOUT	37 (137%)	36 (133%)
37	800	80	15	TIMEOUT	20 (133%)	19 (127%)
38	900	5	29	TIMEOUT	34 (117%)	41 (141%)
39	900	10	23	TIMEOUT	31 (135%)	31 (135%)
40	900	90	13	TIMEOUT	17 (131%)	17 (131%)

Tabela 1 – Resultados