Relatório - Trabalho Prático II

Henrique Oliveira da Cunha Franco
Ciência da Computação
Pontifícia Universidade Católica de Minas Gerais
1448652@sga.pucminas.br

Bernardo Augusto Amorim Vieira
Ciência da Computação
Pontifícia Universidade Católica de Minas Gerais
1449516@sga.pucminas.br

***Resumo*—O problema dos k-centros é uma tarefa clássica na análise de dados e está intimamente relacionado às técnicas de *clustering*. Seu objetivo é identificar k vértices em um grafo completo, minimizando a maior distância de qualquer vértice do grafo ao centro mais próximo. Essa distância é conhecida como o "raio"da solução. O problema possui ampla aplicação em diversos cenários, como categorização de consumidores, alunos ou deputados, e na localização de facilidades, como hospitais ou centros de distribuição.**

Embora a resolução exata seja ideal para pequenas instâncias, ela se torna inviável para problemas maiores devido à sua complexidade combinatória. Assim, abordagens aproximadas ganham destaque por oferecerem soluções aceitáveis com maior eficiência.

I. Introdução

Este artigo visa a implementação e comparação de duas abordagens para resolver o problema dos k-centros: uma abordagem exata e outra aproximada. A primeira garante a solução ótima ao explorar todas as combinações possíveis de k centros, mas é computacionalmente inviável para instâncias grandes. A segunda, baseada em um algoritmo guloso, oferece uma solução aproximada, priorizando eficiência em cenários de maior escala.

Os experimentos utilizam 40 instâncias da *OR-Library*, originalmente criadas para o problema das p-medianas. Cada instância é representada por um grafo completo com custos de aresta e parâmetros como número de vértices ($|V|$), centros (k) e o raio da solução ótima. A análise comparativa entre as duas abordagens avalia a eficácia e eficiência de ambas, especialmente com o aumento do tamanho das instâncias.

Para o pré-processamento, o algoritmo de Floyd-Warshall é empregado para calcular as menores distâncias entre todos os pares de vértices, permitindo o cálculo do raio em ambas as abordagens. A abordagem exata utiliza um método combinatório para testar todas as possíveis combinações de k vértices, enquanto a aproximada seleciona iterativamente os centros que maximizam a distância mínima ao restante do grafo.

Os resultados esperados incluem um melhor desempenho em termos de tempo de execução para a abordagem aproximada em instâncias grandes, enquanto a abordagem exata se destaca em precisão para instâncias pequenas.

Este trabalho não apenas explora a resolução de um problema relevante na ciência da computação, mas também incentiva o desenvolvimento de habilidades analíticas e práticas em algoritmos e estruturas de dados.

II. Método 1: Algoritmo Guloso

Fase de Inicialização

- Crie uma lista vazia para armazenar os centros selecionados:

```
vector<int> centers;
```

- Crie um vetor para rastrear as distâncias mínimas para cada vértice:

```
vector<int> min_distances(n + 1, INT_MAX);
```

Seleção do Primeiro Centro

```
int first_center = 1;
int max_min_dist = 0;
for (int v = 1; v <= n; v++) {
    int min_dist = INT_MAX;
    for (int u = 1; u <= n; u++) {
        min_dist = min(min_dist, dist[v][u]);
    }
    if (min_dist > max_min_dist) {
        max_min_dist = min_dist;
        first_center = v;
    }
}

centers.push_back(first_center);
```

- Encontre um vértice que maximize a distância mínima para os outros vértices.
- Este vértice torna-se o primeiro centro.
- Adicione-o à lista `centers`.

A. Seleção dos Centros Subsequentes

```
while (centers.size() < k) {
    int max_min_distance_vertex = 0;
    int max_min_distance = 0;

    for (int v = 1; v <= n; v++) {
        // Pule se já for um centro
        if (find(centers.begin(), centers.end(), v)
            != centers.end()) continue;

        int min_dist = minDistanceToCenter(v,
            centers);
        if (min_dist > max_min_distance) {
            max_min_distance = min_dist;
            max_min_distance_vertex = v;
        }
    }

    centers.push_back(max_min_distance_vertex);
}
```

Passos-Chave:

- Repita até que k centros sejam selecionados.
- Para cada vértice que não seja centro:
  - Calcule sua distância mínima para os centros existentes.
  - Encontre o vértice com o máximo dessas distâncias mínimas.
- Isso garante que cada novo centro esteja o mais distante possível dos centros existentes.

B. Cálculo do Raio Máximo

```
int max_radius = 0;
for (int v = 1; v <= n; v++) {
    int min_dist = minDistanceToCenter(v, centers);
    max_radius = max(max_radius, min_dist);
}
```

- Calcule a distância máxima de qualquer vértice para o centro mais próximo.
- Isso representa o "raio"da clusterização.

C. Função Auxiliar: Distância Mínima aos Centros

```
int minDistanceToCenter(int vertex, const vector<int
    >& centers) {
    int min_dist = INT_MAX;
    for (int center : centers) {
        min_dist = min(min_dist, dist[vertex][center
            ]);
    }
    return min_dist;
}
```

- Encontra a distância mínima de um vértice para qualquer um dos centros selecionados.

Complexidade do Algoritmo

- **Complexidade de Tempo:** $O(n^2k)$
  - Selecionar centros leva $O(n^2)$ para cada uma das k iterações.
- **Complexidade de Espaço:** $O(n^2)$ devido à matriz de distâncias.

D. Características do Método Guloso

- Não garante encontrar a solução ótima.
- Fornece uma boa aproximação.
- É muito mais rápido que o método exato.
- Funciona bem para grafos grandes.

E. Garantia Teórica

- O algoritmo guloso fornece uma aproximação 2.
- Isso significa que a solução é, no máximo, duas vezes a solução ótima.

F. Visualização do Processo

- Selecione o primeiro centro em uma posição "central".
- Cada centro subsequente é posicionado para maximizar a cobertura.
- Objetivo: Minimizar a distância máxima de qualquer ponto para o centro mais próximo.
- [1] e [2] foram consultados para implementação.

III. Método 2: Obtenção exata para Seleção de k-Centros

A. Introdução ao Pré-Processamento de Distâncias

Para garantir que o cálculo das distâncias entre todos os pares de vértices seja eficiente e preciso, utilizou-se o algoritmo de Floyd-Warshall. Este algoritmo permite calcular as menores distâncias entre todos os pares de vértices em um grafo ponderado. A matriz de distâncias resultante será utilizada como base tanto para o método guloso quanto para o método exato.

Listing 1: Pré-Processamento com Floyd-Warshall

```
Graph(const string& filename) {
    ifstream file(filename);
    if (!file.is_open()) {
        throw runtime_error("Error opening file: " +
            filename);
    }

    file >> n >> m >> k;

    G.resize(n + 1);
    dist.resize(n + 1, vector<int>(n + 1, INT_MAX));

    for (int i = 0; i < m; i++) {
        int v, w, c;
        file >> v >> w >> c;

        G[v].push_back({w, c});
        G[w].push_back({v, c});

        dist[v][w] = c;
        dist[w][v] = c;
    }

    // Inicializa as distàncias próprias como 0
    for (int i = 1; i <= n; i++) {
        dist[i][i] = 0;
    }

    // Algoritmo de Floyd-Warshall
    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (
                dist[i][k] != INT_MAX && dist[k][j]
                    != INT_MAX
                ) {
                    dist[i][j] = min(dist[i][j],
                        dist[i][k] + dist[k][j]);
                }
            }
        }
    }
}
```

A matriz `dist` resultante contém as menores distâncias entre cada par de vértices. Esse vetor de distâncias é essencial para o cálculo do raio máximo no método exato, garantindo que as distâncias sejam pré-computadas para acelerar as etapas subsequentes.

B. Descrição do Método Exato

O método exato utiliza uma abordagem combinatória para explorar todas as possíveis combinações de k centros, garantindo a solução absolutamente ótima. A seguir, o processo é descrito:

C. Estratégia de Geração de Combinações

Listing 2: Código para Geração de Combinações

```
vector<int> all_vertices(n);
iota(all_vertices.begin(), all_vertices.end(), 1);

vector<bool> combination_mask(n, false);
fill(combination_mask.begin(), combination_mask.
    begin() + k, true);
```

- Cria um vetor com todos os índices de vértices.
- Utiliza uma máscara booleana para gerar combinações, iniciando com os k primeiros elementos definidos como `true` e o restante como `false`.

D. Exploração das Combinações

Listing 3: Código para Exploração de Combinações

```
do {
    vector<int> current_centers;
    for (int i = 0; i < n; i++) {
        if (combination_mask[i]) {
            current_centers.push_back(all_vertices[i
                ]);
        }
    }
} while (prev_permutation(combination_mask.begin(),
    combination_mask.end()));
```

- Usa `prev_permutation()` para gerar todas as combinações possíveis de k vértices.
- Para cada iteração, cria um vetor com os vértices que representam os centros potenciais atuais.

E. Cálculo do Raio

Listing 4: Código para Cálculo do Raio

```
int max_radius = 0;
for (int v = 1; v <= n; v++) {
    int min_dist = minDistanceToCenter(v,
        current_centers);
    max_radius = max(max_radius, min_dist);
}
```

- Para cada vértice, calcula a distância mínima até qualquer um dos centros atuais.
- Registra o maior valor dessas distâncias mínimas, que corresponde ao "raio"do cluster atual.

F. Rastreamento da Solução Ótima

Listing 5: Código para Rastreamento da Melhor Solução

```
if (max_radius < min_max_radius) {
    min_max_radius = max_radius;
    best_centers = current_centers;
}
```

- Mantém o controle da combinação com o menor raio máximo.
- Atualiza a melhor solução sempre que uma combinação mais eficiente é encontrada.

G. Análise de Complexidade Computacional

- **Complexidade de Tempo:** $O(\binom{n}{k} \cdot n)$
  - Existem $\binom{n}{k}$ combinações possíveis.
  - Para cada combinação, são necessários $O(n)$ para calcular o raio.
- **Complexidade de Espaço:** $O(n)$, devido ao armazenamento dos vértices e da máscara de combinações.

H. Limitações Práticas

- Funciona bem para grafos pequenos ($n < 15$).
- Torna-se inviável para grafos maiores devido ao crescimento exponencial do número de combinações.

I. Exemplo de Execução

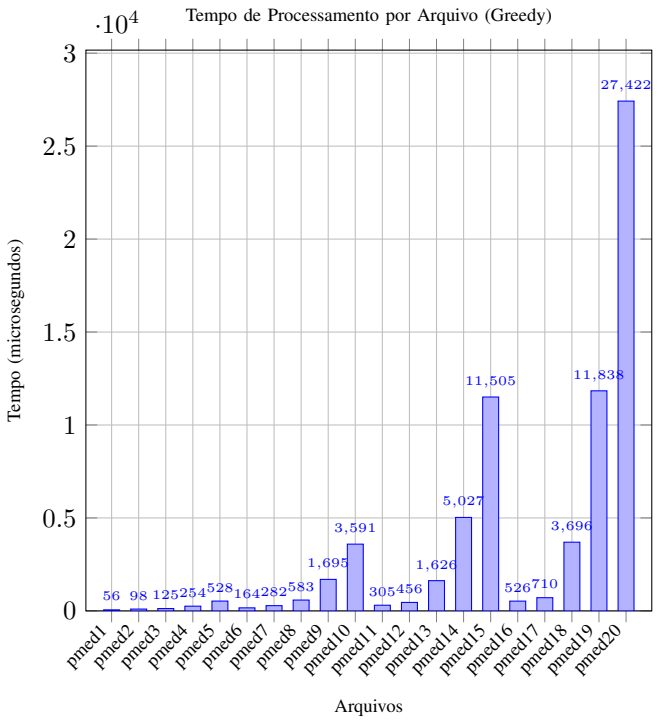
1) Inicia com os k primeiros vértices como centros.
2) Calcula o raio associado a essa combinação.
3) Gera a próxima combinação de k vértices.
4) Calcula o raio da nova combinação.
5) Compara e mantém a combinação com o menor raio máximo.
6) Repete até explorar todas as combinações possíveis.

Diferença para o Método Guloso

Fundamentalmente, a principal diferença em relação ao método guloso é que o método exato garante encontrar a solução absolutamente ótima, pois explora **todas** as combinações possíveis de centros, enquanto o método guloso faz escolhas localmente ótimas.

IV. Resultados

A partir da obtenção dos tempos gastos para cada um dos arquivos utilizando o algoritmo guloso, foi possível elaborar a seguinte tabela:



V. Conclusão

A partir da tabela apresentada, é possível observar que o tempo de processamento do algoritmo guloso de k-centros aumenta significativamente com a complexidade dos arquivos, especialmente à medida que o número de vértices e arestas cresce.

Principais observações:

- Para arquivos menores, como `pmed1` a `pmed8`, o tempo de processamento permanece relativamente baixo, com variações entre 56 e 583 microsegundos. Isso reflete a eficiência do algoritmo em instâncias de menor porte.
- À medida que os arquivos se tornam mais complexos, como `pmed9` a `pmed20`, o tempo de execução cresce exponencialmente, alcançando valores acima de 27.000 microsegundos para `pmed20`. Essa tendência é consistente com a complexidade teórica do algoritmo, que é $O(n^2k)$.
- O crescimento no tempo de processamento é mais acentuado a partir de `pmed13`, onde os tempos saltam de 1.626 microsegundos para mais de 5.000 microsegundos, indicando a sensibilidade do algoritmo ao aumento do número de vértices e arestas.

Implicações:

- Apesar de não garantir a solução ótima, o algoritmo guloso fornece uma solução aceitável em tempos razoáveis para instâncias menores ou moderadas.
- Para instâncias maiores, como aquelas observadas nos arquivos `pmed18` a `pmed20`, o tempo de execução pode se tornar um fator limitante. Nesse caso, soluções mais otimizadas ou paralelizadas podem ser necessárias.

Recomendações:

- Para problemas práticos que envolvam instâncias de grande escala, avaliar a viabilidade de algoritmos alternativos ou estratégias de particionamento do grafo pode ser uma abordagem promissora.
- A implementação do algoritmo pode ser ajustada para reduzir custos computacionais, como a utilização de estruturas de dados mais eficientes para calcular distâncias mínimas.

Em resumo, o algoritmo guloso de k-centros é uma ferramenta eficiente para clusterização, especialmente em casos onde rapidez e simplicidade são priorizadas, mas seus limites de escalabilidade devem ser considerados em aplicações práticas.

Referências

[1] D. Mount, "Cmsc 451: Lecture 8, greedy approximation algorithms: The k-center problem," Lecture notes, 2017, reading: A variant of this problem is discussed in Chapter 11 in KT and Section 9.2.2 in DPV. [Online]. Available: https://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect08-kCenter.pdf

[2] GeeksforGeeks, "Greedy approximate algorithm for k-centers problem," https://www.geeksforgeeks.org/greedy-approximate-algorithm-for-k-centers-problem/, n.d., accessed on: Dec 5, 2024.