UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

ELDER RIBEIRO STORCK - 2020101826 - Eng. de Computação
TALLES CAVALLEIRO WEILER - 2021101923 - Eng. de Computação

AGRUPAMENTO DE ESPAÇAMENTO MÁXIMO

Vitória - ES
2023

ELDER RIBEIRO STORCK - 2020101826 - Eng. de Computação
TALLES CAVALLEIRO WEILER - 2021101923 - Eng. de Computação

AGRUPAMENTO DE ESPAÇAMENTO MÁXIMO

Relatório apresentado à disciplina de Técnicas de Busca e Ordenação como requisito parcial para obtenção de nota.

Professor: Giovanni Ventorim Comarela

Vitória - ES
2023

SUMÁRIO

INTRODUÇÃO

Ao longo deste trabalho será desenvolvido um algoritmo chamado de agrupamento de espaçamento máximo, utilizando uma MST (Minimum Spanning Tree). Inicialmente o trabalho deve receber um conjunto *U*, de n pontos *p1,p2,…,pn*, que para cada par de pontos, *pi* e *pj*, deve ser calculada a distância entre os pontos. O objetivo do trabalho é dividir os pontos do conjunto *U*, em K-grupos não vazios *C1, C2, .., Ck* com base nas distâncias calculadas. Assim os K grupos são definidos com base na distância entre os pontos, de forma que os K-grupos tenham o máximo espaçamento possível.

Para elucidar o tema, será desenvolvido um código em C, que fará a leitura do conjunto *U* de pontos em um arquivo. Após isso, deve-se calcular os K grupos e os registrar em um arquivo de saída, onde cada grupo deve está em uma linha, somando um total de K-linhas. Para uma melhor compreensão do código desenvolvido, foi elaborado o seguinte relatório, onde serão explicadas as principais funções do código e suas funcionalidades, além de relatar o uso do algoritmo de Kruskal, que monta a árvore geradora mínima.

Embora seja relativamente fácil de implementar, o algoritmo de Kruskal possui complexidade de tempo O (m log n), onde m representa o número de arestas e n o número de vértices. Esse tempo de execução é bastante eficiente para grafos com um número moderado de arestas, mas pode ser muito lento para grafos muito grandes. Existem, no entanto, técnicas para acelerar o desempenho do algoritmo em alguns casos, tornando-o uma ferramenta ainda mais útil para a resolução de problemas em diversas áreas.

METODOLOGIA

Descrição das principais decisões de implementação, incluindo uma justificativa para os algoritmos e estruturas de dados escolhidos.

PRINCIPAIS FUNÇÕES:

FILE* utility_openFile(char *nameFile, char *action)

faz a abertura correta do arquivo no modo de leitura e retorna um ponteiro para o arquivo

int utility_amountLine(FILE *file)

determina a quantidade de pontos que um arquivo tem, retornando exatamente a quantidade de pontos, sendo um número inteiro.

int utility_amountLine(FILE *file) e int utility_amountToken(FILE *file)

determina a quantidade de coordenadas de um ponto, retornando a quantidade de tokens, que é a quantidade de coordenadas de um ponto acrescido de 1, sendo um número inteiro.

Point* Point_readFile(FILE *file, int amountPoints, int amountCoordinates)

Faz a leitura dos pontos contidos no arquivo, para isso além de ler o arquivo, aloca os dados dos pontos em um vetor de “struct Point” (estrutura criada para o armazenamento do nome, do ID e das coordenadas dos pontos disponibilizados). retornando uma lista com todas as informações de um ponto.

Edge* Edge_initAndCaculate(Point *list_points, int amountPoints, int amountCoordinates, int amountEdge)

Faz o cálculo das arestas entre os pontos, Além de calcular a distância, aloca os dados das arestas em um vetor de “struct Edge” (estrutura criada para o armazenamento o ID do ponto 1, o ID do ponto 2 e da distância entre eles). retornando uma lista com todas as arestas alinhadas por menor distância.

Tree* monta_arvore(Edge* list_edge, Point *list_points, int k, int amountPoints, int amountEdge)

Monta uma árvore geradora mínima com (k-1) separações. Além de montar a árvore geradora mínima, aloca os dados dos pontos em um vetor de “struct Tree” (estrutura criada para o armazenamento o ID, o nome e o tamanho do nó que representa o ponto). retornando uma lista com todos os pontos alinhando pelos grupos de pais em comum.

void outFile_write(Point *list_points, Tree *list_tree, char *nameFile, int amountPontos, int k)

Escreve no arquivo de saída os grupos separados por linha e por ordem lexicográfica. Faz a ordenação dos grupos por nome e gera o arquivo de saída com os dados dos grupos.

void Edge_free(Edge *list_edge), void Point_free(Point *list_points, int amountPoints) e void UF_free(Tree *list_tree, int size)

Funções que servem para dar free nos mallocs e nas estruturas alocadas para o funcionamento do código.

ESTRUTURAS USADAS:

typedef struct point Point

Estrutura criada para armazenar um ponteiro para o nome, id e as coordenadas do ponto.

```
struct point{
    char  *name;
    int      id;
    float *coordinates;
};
```

typedef struct edge Edge

Estrutura que serve para armazenar dois pontos (através id) e o resultado da distância entre eles.

```
struct edge{
    int   ID1;
    int   ID2;
    float distance;
};
```

typedef struct tree Tree

Estrutura que armazena a posição do pai, o nome dele e o tamanho da árvore.

```
struct tree{
    /* data */
    int     id;
    char *name;
    int     sz;
};
```

COMPLEXIDADE DO CÓDIGO

A seguir será calculado a complexidade do código a partir das seguintes funções:

int utility_amountLine(FILE *file)

A função apresentada tem uma complexidade temporal de O(N), onde N é o número de linhas no arquivo. Isso ocorre porque o código percorre o arquivo inteiro uma única vez para contar o número de linhas, sem realizar operações adicionais que dependam do número de linhas.

int utility_amountToken(FILE *file)

A função apresentada tem uma complexidade temporal de O(N), onde N é o número de tokens na primeira linha do arquivo. Isso ocorre porque o código percorre a primeira linha do arquivo uma única vez para contar o número de tokens separados por vírgulas.

Point* Point_readFile(FILE *file, int amountPoints, int amountCoordinates)

A complexidade dessa função depende do número de linhas no arquivo de entrada e do número de coordenadas por ponto. Dentro do loop while, a operação getline() é chamada para cada linha do arquivo, o que leva tempo O(n), onde n é o número de linhas do arquivo. Dentro do loop do-while interno, a operação strtok() é chamada para cada coordenada em uma linha, o que leva tempo O(m), onde m é o número de coordenadas em uma linha. O tempo de execução total do código é, portanto, O(n*m), que é a complexidade assintótica do algoritmo.

Edge* Edge_initAndCaculate(Point *list_points, int amountPoints, int amountCoordinates, int amountEdge)

A complexidade da função é O(n^2 * m * log m), onde n é o número de pontos e m é o número de coordenadas. O loop externo for percorre a lista de pontos, o que leva tempo O(n). O segundo loop for percorre os pontos restantes na lista de pontos, o que leva tempo O(n) novamente. O terceiro loop for percorre as coordenadas dos pontos, o que leva tempo O(m). Em geral, o tempo de execução do código é dominado pelo qsort(), que leva tempo O(m * log m) para ordenar as distâncias entre os pontos. Portanto, a complexidade assintótica total do código é O(n^2 * m * log m).

Tree* monta_arvore(Edge* list_edge, Point *list_points, int k, int amountPoints, int amountEdge)

A complexidade da função monta_arvore depende do número de pontos e arestas no grafo, bem como do valor de k. O loop for que percorre o vetor de arestas é executado no máximo amountEdge vezes, onde amountEdge é o número de arestas no grafo. No entanto, ele pode ser interrompido prematuramente se count atingir o valor amountPoints - k. Portanto, a

complexidade desse loop é O(min(amountEdge, amountPoints-k)), ou simplesmente O(amountEdge) se amountEdge for menor que amountPoints - k. A chamada para qsort tem complexidade O(n * log n), onde n é a quantidade de pontos no grafo. Portanto, a complexidade total da função monta_arvore é de O(n * log n) pois é a de maior complexidade.

void outFile_write(Point *list_points, Tree *list_tree, char *nameFile, int amountPontos, int k)

A complexidade dessa função depende de n e k, onde n é a quantidade de pontos e k é a quantidade de grupos desejada.A primeira parte do código é o loop que ordena cada grupo de pontos separadamente, que tem complexidade O(k * (n/k) * log(n/k)), pois cada grupo tem tamanho n/k e a ordenação leva log(n/k) tempo. A segunda parte é o loop que cria a lista de grupos de árvores e a lista de primeiros nós de cada grupo. Esse loop tem complexidade O(k * (n/k)) para a criação de cada lista, e a ordenação subsequente com a função ordenaPorName tem complexidade O(k * (n/k) * log(n/k)). A terceira parte é o loop que percorre cada grupo e escreve os nomes dos pontos no arquivo. Esse loop tem complexidade O(k * (n/k)). Portanto, a complexidade total é O(k * (n/k) * log(n/k) + k * (n/k)) + O(k * (n/k) * log(n/k)) + O(k * (n/k)) + O(k * (n/k)) = O(k * (n/k) * log(n/k) + k * (n/k)). Como queremos a maior complexidade, logo temos O(k * n * log n).

void Point_free(Point *list_points, int amountPoints)

Esta função tem complexidade O(n), onde n é a quantidade de pontos na lista. Isso ocorre porque ela itera sobre cada ponto na lista (n iterações) e libera a memória alocada para as coordenadas e para o nome do ponto, além de liberar a memória alocada para a lista de pontos. Como cada operação de liberação de memória tem complexidade constante, a complexidade total da função é O(n).

ANÁLISE EMPÍRICA

O programa passou por uma série de testes fornecidos pelo professor para realizar a análise. Durante esse processo, foram medidos os tempos (em segundos) necessários para ler os dados, calcular as distâncias, ordená-las, obter a árvore geradora mínima (MST), identificar os grupos e escrever o arquivo de saída.

	1	2	3	4	5
Leitura dos dados	0,000057	0,000103	0,00287	0,014721	0,011596
Calculo das distâncias	0,000116	0,000548	0,038669	0,527686	4,033113
Ordenação das distâncias	0,000173	0,000733	0,111914	0,787530	3,397528
Obtenção da mst	0,000011	0,00002	0,001211	0,004344	0,010118
Identificação dos grupos	0,000009	0,000014	0,000171	0,000505	0,000986
Escrita do arquivo de saída	0,000123	0,000125	0,000327	0,000753	0,001379
Soma	0,000489	0,001543	0,155162	1,335539	7,454720

Porcentagens:

	1	2	3	4	5
Leitura dos dados	11,66%	6,68%	1,85%	1,10%	0,16%
Calculo das distâncias	23,72%	35,52%	24,92%	39,51%	54,10%
Ordenação das distâncias	35,38%	47,50%	72,13%	58,97%	45,58%
Obtenção da mst	2,25%	1,30%	0,78%	0,33%	0,14%
Identificação dos grupos	1,84%	0,91%	0,11%	0,04%	0,01%
Escrita do arquivo de saída	25,15%	8,10%	0,21%	0,06%	0,02%
Soma	100,00%	100,00%	100,00%	100,00%	100,00%

Com base na análise, fica visível que as funções que mais custam tempo são as de calcular as distâncias e ordenação das distâncias, o que faz sentido contando que são a parte principal do código