Técnicas de Busca e Ordenação

Trabalho Prático I - Problema de Agrupamento

Aluno: Nathan Garcia Freitas / Matrícula: 2022102179

Professor: Giovanni Ventorim Comarela

1- Introdução

O trabalho foi realizado com o intuito de resolver um problema de agrupamento de pontos no espaço. Ao decorrer do trabalho, foi utilizado algoritmos para realizar a criação de uma Árvore Geradora Mínima, e para unir os pontos, baseado nessa árvore. Os testes realizados foram com base nos exemplos cedidos pelo professor, e o tempo limite de execução (sem executar o programa com o valgrind) foi de 15s.

2- Metodologia

Foram criadas 4 TAD's no trabalho, 2 referenciando um ponto (Point e PointList) e 2 referenciando uma aresta (Edge e EdgeList). Além dessas estruturas, houve a criação de arquivos separados para as funções de UF_find e UF_union (union.c e union.h) além de um arquivo para os algoritmos utilizados ao longo do trabalho (algorithm.c e algorithm.h).

Iniciando pelas TAD's referentes aos pontos, a estrutura Point armazena: uma string com a ID do ponto, um vetor com todas as coordenadas do ponto, além de variáveis inteiras que armazenam seu grupo, a dimensão do ponto (tamanho utilizado do vetor de coordenadas) e a quantidade de memória alocada para o vetor de coordenadas. Já PointList armazena um vetor de ponteiros para a TAD Point, junto de duas variáveis inteiras que tem o tamanho alocado para esse vetor e a quantidade de pontos que tem no vetor.

As TAD's de arestas ao invés de armazenar os pontos propriamente, armazenam seus índices no vetor de pontos, visando mais simplicidade na implementação do código, além claro de armazenar o valor da distância euclidiana entre os dois pontos (weight). A EdgeList possui um vetor de ponteiros para arestas, e possui também a quantia de arestas no vetor, além de seu tamanho alocado, que é calculado pela fórmula:

$size = (n^2 / 2) - (n / 2)$, sendo n o número de pontos lidos.

Os algoritmos utilizados foram o kruskal e o cluster, algoritmos recomendados nas especificações do trabalho que seriam os melhores para o nosso caso, onde ambos os algoritmos possuem a utilização do UF. A função **Kruskal** é responsável por criar a MST (Árvore Geradora Mínima), que também é uma EdgeList, utilizando a EdgeList, já ordenada, criada com os pontos lidos do arquivo de entrada, unindo os pontos das arestas e colocando essas arestas na MST, sem criar cópias, apenas passando o endereço dessas arestas para a MST, inclusive esse algoritmo é o responsável por evitar as redundâncias nas ligações entre as arestas. Enquanto isso, a função **Group Designation** é responsável por unir os pontos da PointList lida do arquivo de entrada em um mesmo grupo, baseando-se nos pontos das arestas inseridas na MST, valendo ressaltar que ambas funções criam vetores de inteiros para representar os pontos, com isso a representação dos pontos nas arestas como seus índices facilita o funcionamento dos algoritmos de UF.

Falando a respeito do UF, foram utilizadas as funções UF_find e UF_union (adaptadas como find() e unite()). Em ambos os casos, a representação dos pontos dentro de uma aresta ser através de seus índices no vetor de pontos facilitou a utilização das funções propostas pelo professor em sala, tornando a adaptação para o trabalho mais simples, aliás, já falando a respeito de complexidade, a versão do algoritmo UF utilizada foi a **Weighted QU + path compression,** que apresenta menores tempos de execução que outras versões do próprio algoritmo.

3- Análise de Complexidade

Leitura: A função de leitura "points_reader" lê todas as informações dos pontos no arquivo de entrada. Sendo *N* = quantidade de pontos; e *M* = dimensões do ponto + 1 (onde esse 1 representa o ID do ponto), temos a complexidade da leitura de *M x N*.

Cálculo das Distâncias: Para calcular todas as arestas utilizadas em um grupo de N pontos, sem que haja redundâncias (aresta de A com A, ou até mesmo repetição de arestas, como a existência de A-B e B-A), há uma fórmula, onde temos *N* sendo o número de pontos, temos que o total é: $(n^2 / 2) - (n / 2)$, com isso, sabemos que a complexidade é: N^2.

Ordenação das Distâncias: Para esse trecho foi utilizada a função qsort, uma função de ordenação de uma biblioteca C, que possui complexidade: $N.log_2 N$.

Gerar MST: Para gerar a MST foi utilizado o algoritmo de Kruskal, com o UF Weighted Quick-Union com compressão de caminho, portanto a complexidade é $N + N.M.lg * N$ onde *N* é a quantidade de arestas e *M* é a quantidade de operações de UF feitas. Temos esse N somado devido aos vetores criados para auxiliar no funcionamento das funções de UF.

Identificar Grupos: $N.M.lg * N + 2N$, visto que a complexidade é igual aos do Kruskal, porém ao invés de *N*, será *2N*, pois teremos a atualização dos grupos no vetor de pontos.

Escrita Saída: $N^2/2$**,** um loop para passar por todos os pontos, verificando o grupo do ponto atual, e com o grupo obtido, passando por todos os demais pontos a seguir que possuem o mesmo grupo, imprimindo-os.

4- Análise Empírica

Tabela de Tempo Total da Execução

Casos	Tempo Total	Leitura	Cálculo Distâncias	Ordenação Distâncias	Gerar MST	Identificar Grupos	Escrita Saída
1.txt	0.000674 s	≈ 10,70%	≈ 17,0%	≈ 32,22	≈ 9,51%	≈ 1,63%	≈ 22,40%
2.txt	0.001918 s	≈ 6,51%	≈ 18,56%	≈ 49,32%	≈ 10,21%	≈ 1,93%	≈ 8,41%
3.txt	0.236969 s	≈ 0,24%	≈ 9,96%	≈ 51,16%	≈ 15,52%	≈ 0,07%	≈ 0,16%
4.txt	2.337489 s	≈ 0,11%	≈ 8,05%	≈ 43,88%	≈12,58%	≈ 0,02%	≈ 0,04%
5.txt	9.046303 s	≈ 0,10%	≈ 9,96%	≈ 56,67%	≈ 15,24%	≈ 0,01%	≈ 0,02%

O tempo total considera o tempo para desalocação de memória, e os casos testes utilizados são os fornecidos pelo professor, sendo os casos:
1: n = 50; k = 2; m = 2;
2: n = 100; k = 4; m = 3;
3: n = 1000; k = 5; m = 2;
4: n = 2500; k = 5; m = 5;
5: n = 5000; k = 10; m = 10;