Grupo: Ana Tereza Pereira, Felipe Albuquerque e Vitor Dadalto
Disciplina: Técnicas de Busca e Ordenação
Professor: Giovanni Comarela

Relatório

Trabalho Prático TI

17 de abril de 2024, Vitória - ES

Sumário

Introdução

O Agrupamento de Espaçamento Máximo é um problema clássico da computação e sua solução é o objetivo deste projeto. Nesse documento, detalharemos a metodologia utilizada para alcançarmos esse resultado e analisaremos o desempenho dessa aplicação quanto à sua complexidade e empiricamente.

Metodologia

O projeto foi desenvolvido utilizando uma struct principal Cluster, que encapsula todas as funções e os outros tipos, assim sendo a única TAD que se comunica com a `main()`. Optamos por utilizar apenas arrays dentro dos tipos como estrutura de dados devido ao acesso às informações ter complexidade linear e à maior facilidade na utilização de funções nativas do C como o `qsort()`. Na etapa do algoritmo de Kruskal (Utilizado para gerar a Minimum Spanning Tree), utilizamos o algoritmo Weighted Quick Union devido ao seu desempenho otimizado em relação ao Quick Find e ao Quick Union e à simplicidade na sua implementação e na codificação entre pais e filhos, armazenados em um array que representa a árvore.

Análise de complexidade

Legenda:

K = número de grupos;

M = número de dimensões;

N = Número de pontos.

1. **Leitura dos dados** (`cluster_read()`)

Caso	Notação ~	Notação O
Melhor	$\sim(N \times M)$	O(N)
Pior	$\sim(N \times M)$	O(N)

É necessário passar por cada linha da entrada, e cada linha contém as informações de 1 ponto, logo é necessário passar por N linhas. Mas cada linha contém as M coordenadas do ponto. Foi considerado que N >>> M para atribuir a complexidade.

2. **Cálculo das distâncias** (`cluster_calcDistances()`)

Caso	Notação ~	Notação O
Melhor	$\sim(N^2 \times M)$	$O(N^2)$
Pior	$\sim(N^2 \times M)$	$O(N^2)$

É preciso calcular a distância entre cada par de pontos, que é o resultado de uma combinação $C_{n,2} = \frac{N \times (N-1)}{2}$, e para calcular cada distância é preciso iterar pela diferença entre as M coordenadas de cada ponto.

3. **Ordenação das distâncias** (`cluster_sortDistances()`)

Caso	Notação ~	Notação O
Melhor	$\sim(D \times log(D))$	$O(D \times log(D))$

Pior	$\sim(D^2)$	$O(D^2)$

É somente realizado um qsort() no vetor de Distâncias com tamanho $D = (N^2 - N)/2)$. Logo, a complexidade da função é a mesma do qsort()

4. **Obtenção da MST** (`cluster_kruskal()`)

Caso	Notação ~	Notação O
Melhor	$\sim((N + D)lgN + N))$	$O(DlgN))$
Pior	$\sim((N + D)lgN + N))$	$O(DlgN))$

É utilizado o algoritmo Weighted Quick Union. Ao início, são inicializadas as N posições do array que guarda o tamanho de cada sub-árvore com 1. Após isso, ocorre a iteração por cada distância, e em cada iteração é buscada a raiz da sub-árvore do ponto A e do ponto B (essa busca pela raiz na WQU possui complexidade $lg(N)$). Durante esse loop, o if que contém a união de duas sub-árvores (com complexidade de $lg(N)$) é acessado (N - 1) vezes.

5. **Identificação dos grupos** (`cluster_identifyGroups()`)

Caso	Notação ~	Notação O
Melhor	$\sim(N)$	$O(N)$
Pior	$\sim(N^3 - N^2 + N)$	$O(N^3)$

No melhor caso, em que k = N, basta percorrer o array de pontos (que possui tamanho N) e ir adicionando 1 ponto em cada grupo. Já no pior caso, é necessário percorrer o array de pontos e atribuir cada ponto a um grupo. Após isso, é preciso fazer merge de todos os grupos, para isso, é preciso iterar sobre o array de arestas (que possui tamanho N - 1) e, para cada iteração, obter o grupo do ponto A e do ponto B (passando 2 vezes por uma função com

complexidade $N(N-1)$, no pior caso) e realizar o merge desses grupos (que possui complexidade $(N-1)$, no pior caso).

6. **Geração do arquivo de saída** (`cluster_generateResult()`)

Caso	Notação ~	Notação O
Melhor	$\sim(N + Nlog(N))$	$O(Nlog(N))$
Pior	$\sim(N + Nlog(N))$	$O(Nlog(N))$

É necessário identificar os grupos que, ao final da sequência de merges, não ficaram vazios, para isso é necessário iterar sobre N grupos. Após isso, é preciso ordenar cada grupo em ordem lexicográfica, utilizando o qsort() e após isso ordenar o array de grupos utilizando o qsort() novamente. Por fim, é necessário passar por cada grupo e iterar sobre seus pontos, imprimindo-os. Note que em um dos casos extremos existe 1 grupo com N elementos e no outro existem N grupos com 1 elemento, que resultam em complexidades semelhantes.

Análise empírica

Essa análise baseou-se numa média dos tempos de execução dos computadores dos 3 integrantes do grupo, descrita na tabela a seguir, para cada entrada fornecida.

	n	m	k	distArraySize
ex1	50	2	2	1225
ex2	100	3	4	4950
ex3	1000	2	5	499500
ex4	2500	5	5	3123750
ex5	5000	10	10	12497500

	cluster_read()	cluster_calcDistances()	cluster_sortDistances()	cluster_kruskal()
ex1	0	0	0	0
ex2	0,00033	0	0,00033	0
ex3	0,002	0,025	0,10566	0,061
ex4	0,01	0,16667	0,847	0,482
ex5	0,025	0,84534	4,08966	2,066

	cluster_identifyGroups()	cluster_generateResult()	cluster_destroy()	total time
ex1	0	0	0	0,00066
ex2	0	0,00033	0	0,00133
ex3	0,056	0,002	0	0,25266
ex4	0,54266	0,003	0,001	2,05566
ex5	2,547	0,004	0,0025	9,582

Como podemos observar, os tempos de execução das funções que tiveram pior complexidade na análise de complexidade cresceram mais rapidamente com o aumento da quantidade de pontos (N), como `cluster_identifyGroups()` e `cluster_sortDistances()`, assim confirmando nossa análise prévia.