UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CENTRO TECNOLÓGICO

ESTRUTURA DE DADOS 2

Beatriz Maia

Ryan Monteiro

Sophie Dilhon

Trabalho Prático 1: Agrupamento de Espaçamento Máximo

Vitória, ES

2021

Sumário

1 Introdução

1.1 Resumo

Este trabalho visa resolver o problema do agrupamento de espaçamento máximo, criando grupos com alguma similaridade. O algoritmo busca criar k grupos de pontos presentes no plano R^m, com base em suas distâncias, por meio da criação de uma árvore geradora mínima usando o algoritmo de Kruskal (KRUSKAL, 1956).

Este documento tem como objetivo relatar o algoritmo e estruturas, implementadas na linguagem C, no trabalho 1 de Estruturas de Dados II, Foram feitas analises, dos algoritmos usados, de complexidade e empíricamente com base no tempo de execução do programa.

1.2 Propriedades da máquina

Os testes foram rodados em um computador com as seguintes especificações:

Processador: AMD Ryzen 5 2400G 3,60 GHz

Memória RAM: 2x8GB 2400MHz

2 Metodologia

Foram implementados os arquivos distancia, planoR, ponto, pilha, uf, e leArquivo, tendo os cinco primeiros as estruturas usadas para o algoritmo de espaçamento máximo. As métricas observadas foram memória alocada e tempo necessário para rodar o programa.

2.1 Estruturas

2.1.1 Ponto

A estrutura ponto é composta por um vetor de char referente ao id e um vetor de double contendo suas coordenadas. Ela também possui um atributo denominado raiz, que recebe o valor do índice do ponto no vetor de pontos.

A escolha dessa estrutura se deu pensando na simplicidade do ponto, já o uso da raiz dentro do ponto, para auxílio na estrutura *union-find*, que explicaremos mais a frente.

2.1.2 Pilha

A estrutura pilha é composta por: um inteiro referente a quantidade de pontos, e outro a dimensão dos pontos, além de uma lista encadeada de pontos sem sentinela.

A pilha é usada apenas na leitura do arquivo, dada a quantidade de pontos ser desconhecida. A decisão do uso da pilha com lista encadeada contra o uso do *realloc* é dado a variação do valor N em diferentes casos, podendo ser na escala de unidades ou na escala dos milhares. Com isso a dificuldade de escolha de um tamanho para cada alocação de memória, que poderia resultar em um tamanho muito pequeno para o caso, necessitando de chamadas repetidas da função, ou em um tamanho muito grande para o caso.

2.1.3 Plano

O planoR possui dois inteiros, sendo eles, a dimensão dos pontos e a quantidade de pontos, e um vetor de pontos preenchido com auxílio da pilha explicada na seção anterior.

A escolha da estrutura planoR se deu ao fato do acesso a um ponto em uma lista encadeada, presente na pilha, ser mais custoso comparado a mesma operação em um vetor, em que basta acessar a posição desejada através do valor do índice. Além de ser necessário ordenar os pontos lexicograficamente, algo complexo em uma lista encadeada, portanto foi implementado o planoR.

2.1.4 Distância

A estrutura Distancia contém dois ponteiro para pontos (presentes no vetor de pontos do planoR) e a distância entre eles calculada pela Equação 2.1 abaixo, sendo m a dimensão do ponto:

$$\sqrt{\sum_{i=1}^{m}(x(i) - y(i))^2} \tag{2.1}$$

A partir disso, é criado um vetor em que cada posição aponta para um TAD Distancia, que possui tamanho conforme a Equação 2.2.

$$\frac{(n-1) * n}{2} \tag{2.2}$$

Esse valor é calculado a partir da observação de uma progressão aritmética quando distribuído as distâncias entre os pontos em uma matriz 2D, como na Figura 1. Na Seção 3.2, o cálculo e a complexidade do tamanho é abordado de forma mais completa.

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
A										
B	5.39									
C	5.66	6.08								
D	1.00	5.83	5.00							
E	6.32	1.00	6.32	6.71						
F	4.24	5.39	1.41	3.61	5.83					
G	1.41	5.00	4.24	1.00	5.83	2.83				
H	6.08	1.41	5.39	6.32	1.00	5.00	5.39			
I	5.00	5.10	1.00	4.47	5.39	1.00	3.61	4.47		
J	5.00	6.32	1.00	4.24	6.71	1.00	3.61	5.83	1.41	

Figura 1 – Matriz de todas distâncias calculadas entre 10 pontos. Fonte: Enunciado do Trabalho Pratico T1

O uso do vetor se deu pela facilidade de ordená-lo e de acessar tanto a distância quanto os pontos presentes em cada posição do vetor (comparado a uma lista encadeada).

2.1.5 *Union-find*

Por fim, foi implementada a estrutura **uf** que contém o número de pontos, um ponteiro para o vetor de pontos do **planoR**, e um vetor de inteiro referente ao tamanho das "árvores" (número de conexões). Inicialmente as raízes dos nós referem-se as posições dos pontos no vetor de pontos do plano R, então usando o vetor de Distancia, na função *agrupaCaminhos*, são criadas as $n - k$ uniões entre os nós, e são atualizadas as raízes e tamanhos.

Visando facilitar a identificação de cada grupo, é feito *path compression* na busca da raiz, atribuindo esta ao nó avô, e após as n-k uniões é feita uma busca de forma a forçar que todos os tamanhos sejam de no máximo um.

Essa estrutura foi feita baseada no algoritmo *Weighted Quick Union with Path Compression* (SEDGEWICK; WAYNE, 2011).

2.2 Algoritmos

2.2.1 Leitura e criação das estruturas pilha e plano

Inicialmente faz se a leitura do arquivo de entrada na função leArquivo, lê-se a primeira linha para descobrir a dimensão ponto (M), e novamente é feita a leitura da primeira linha (fechando e abrindo o arquivo novamente) para então alocar as coordenadas, o id, a estrutura ponto, e inseri-lo na pilha. Em seguida, são lidos os outros pontos, que são alocados e também inseridos na pilha, ao longo da leitura é contado o número de pontos N (número de linhas), que é armazenado. Então é alocado um vetor com o valor N, armazenado na pilha, e são inseridos os pontos também presentes na mesma, este vetor esta contido na estrutura PlanoR. Por fim a pilha é liberada.

2.2.2 Agrupamento

A estrutura PlanoR é usada para criar o vetor de Distancias (cada posição do vetor aponta para uma estrutura Distancia), como mostrado na Seção 2.1.4. O vetor de Distancias possui tamanho conforme a Equação 2.2, e é preenchido com as distâncias entre os N pontos seguindo a Equação 2.1, por questões de custo, não é feita a raiz do cálculo, uma vez que já é possível comparar os valores obtidos. O vetor é então ordenado crescentemente com base nas distâncias entre os pontos pela função *qsort* (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2011a).

Enfim é alocada a estrutura UF, o vetor de tamanhos desta é todo preenchido com o valor 1. Como o vetor de distâncias está ordenado, são feitas n-k uniões entre as menores arestas, para obter os k grupos. Primeiro busca-se a raiz de cada nó (ponto), na busca é feito o *path compression* (raiz do nó passa a ser o avó), em seguida a união é feita mudando a raiz da menor árvore para o nó da maior a quem este está sendo conectado, e então os tamanhos das árvores são incrementadas. Após as uniões serem feitas, todos os nós rodam a busca de raiz, forçando que exceto as k raízes, todos os outros nós não tenham filhos.

2.2.3 Escrita dos grupos

Com os grupos criados, é necessário imprimi-los no arquivo de saída, para isto, o vetor de pontos é ordenado lexicograficamente, usando a função *strcmp* (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2011b) como comparação, e a função *qsort* para a ordenação. Então se conferem todas as raízes dos pontos, e são procurados todos os pontos com a mesma raiz e impressos em uma mesma linha, após ser conferida

a raiz passa a ter valor -1. A cada raiz de valor novo, um contador de valor inicial K (a quantidade de grupos existente) é reduzido 1.

3 Análise de complexidade

Para a análise de complexidade, será utilizado o modelo de custo, considerando uma operação básica, como o acesso em *array* ou comparações, como representante do tempo de execução. Todos os cálculos serão simplificados para determinar a ordem de crescimento, e para representa-lá será utilizada a notação *Big O*.

3.1 Leitura e criação do plano

De acordo com a Seção 2.2, na leitura de arquivo há a criação de N pontos com M coordenadas e sua inserção numa pilha de lista encadeada [2.1.2]. Portanto, o crescimento da criação de pontos é igual a $f(N, M)$, como mostrado na Equação 3.1. Para os casos utilizados durante a implementação do código, houve a relação $M <<< N$, apresentando um crescimento linear. Porém, a relação observada não será necessariamente verídica para todos os casos existentes, em que se M tiver tamanho igual a N, a ordem de crescimento é quadrática e assim em diante. Como há uma variação do valor de M, para representação da complexidade de criação dos pontos vamos utilizar a notação $O(N * M)$.

$$f(N, M) = N * M \tag{3.1}$$

A ordem de crescimento de inserção de pontos à pilha, em relação a acesso a *array*, é $O(1)$ visto que há a inserção no topo da pilha, sem precisar percorrer todos os elementos.

Na criação da estrutura planoR [2.1.3], para cada posição do vetor *pontos*, é atribuído um ponto, resultando em N acessos ao *array*. De tal forma, a ordem de crescimento é linear, representada pela Equação 3.2 e a notação $O(N)$.

$$g(N, M) = N \tag{3.2}$$

A complexidade de leitura e criação do plano pode ser escrita como na Equação 3.3 e para representar sua ordem de crescimento podemos utilizar a notação $O(N * M)$.

$$\begin{aligned} C(N, M) &= f(N, M) + g(N, M) \\ C(N, M) &= N(M + 1) \\ C(N, M) &=\sim N * M \end{aligned} \tag{3.3}$$

3.2 Cálculo de distâncias

Para calcular as distâncias, é preciso saber a quantidade de distâncias a ser calculada. Como dito na Seção 2.1.4, podemos utilizar uma matriz triangular 2D para representação de todas as distâncias possíveis entre os pontos. Cada linha (representada por i_0 a i_N) e coluna (representada por j_0 a j_N) indica um ponto dos N pontos, e os valores na $matriz[i_x][j_y]$ representam a distância entre o ponto i_x e o ponto j_y

Utilizando a Figura 1, com *N=10*, é possível observar que a quantidade de distâncias calculadas na coluna A (as distâncias entre A e todos os outros pontos) é *N-1*. Na coluna seguinte, há *N-2* distâncias. Na coluna J, o enésimo ponto, há 0 distâncias. É possível observar uma progressão aritmética entre as quantidades de distâncias nas colunas. A soma da progressão aritmética é representada pela Equação 3.4.

$$\begin{aligned} h(N) &= \frac{N(N-1)}{2} \\ h(N) &= \frac{N^2 - N}{2} \\ h(N) &=\sim N^2 \end{aligned} \tag{3.4}$$

Na implementação do código, o vetor é inicializado com o tamanho $g(N)$ e para cada posição do vetor é atribuído uma distância, tendo uma ordem de crescimento $O(N^2)$.

3.3 Criação das arestas

Na função *agrupaCaminhos*, são criadas as arestas entre os pontos de menor distância, entre N distâncias. O algoritmo permite a criação de K grupos. No pior caso, o vetor é percorrido $N-1$ vezes, criando o agrupamento K apenas no último acesso ao vetor de Distancia. Para verificar se há conexão entre os pontos e fazer as conexões caso não haja caminho, é utilizado o *union-find*, que possui complexidade $O(lgN)$. A complexidade da criação das aresta pelo menor caminho possível é $O(N * lgN)$, visto que o vetor Distancia é percorrido apenas uma vez.

3.4 Ordenações

A ordenação é feita utilizando a função *qsort*, da biblioteca padrão *stdlib.h*, que implementa o algoritimo *quick sort*. A documentação da implementação da função não informa a complexidade da função, porém na literatura (SEDGEWICK, 1978) a complexidade do pior caso é de $O(N^2)$.

3.5 Identificação e escrita dos grupos

A identificação do grupo é dividido em duas partes: uma parte descrita pela Seção 2.2.2 e outra parte descrita pela Seção 2.2.3. Após criadas as arestas, é percorrido novamente o vetor de N pontos para garantir que todos elementos de um grupo (que estão conectados) possuem o mesmo pai, utilizando o método *find* para verificação de caminho, que apresenta complexidade $O(lgN)$, resultando numa ordem de crescimento final $O(N * lgN)$. Nessa etapa os grupos já estão formados, tendo apenas K raízes, porém estão desordenados no vetor.

Na escrita dos grupos, o vetor é percorrido no pior caso N vezes, e no melhor caso K vezes, para achar o pai do nó. Após obtenção do valor do pai, é procurado no vetor, iniciando do ponto seguinte ao de que foi obtido a raiz, se há um ponto com o mesmo pai. De tal forma como o cálculo de distâncias, resultando numa complexidade de $O(N^2)$.

4 Análise empírica

Para a análise empírica, iremos nos basear em testes executados na máquina apresentada na Seção 1.2.

A primeira tabela apresenta os valores de elementos (N), quantidade de coordenadas de cada elemento (M) e a quantidade de grupos finais (K) de cada Arquivo de Entrada usado nos experimentos.

Arquivo	N	M	K
0.txt	10	2	3
1.txt	50	2	2
2.txt	100	3	4
3.txt	1000	2	5
4.txt	2500	5	5
5.txt	5000	10	10

Tabela 1 – Valores de N, M e K de cada arquivo de entrada

A tabela a seguir [2] apresenta os valores de tempo que cada parte do algoritmo leva para ser executado em relação aos diferentes arquivos de entrada, que possuem quantidades diferentes de elementos. Ela apresenta também a porcentagem de tempo de cada operação em relação ao tempo total do algoritmo.

Esse tempo foi obtido com a utilização da função *clock()* e o tipo *clock_t* que são capazes de representar o tempo que o processador usa, a partir do instante que a linha com esse comando for chamada.

Tempo								
Arquivo	Leitura dos dados	Cálculo das distâncias	Ordenação das arestas	Obtenção da MST	Identificação dos grupos	Escrita do arquivo de saída	Liberação das Estruturas	Total
0.txt	0.001272s	0.000003s	0.000004s	0.000001s	0.000001s	0.000415s	0.000002s	0.001698s
	74.91%	0.18%	0.24%	0.06%	0.06%	24.44%	0.12%	100.00%
1.txt	0.000280s	0.000242s	0.000399s	0.000040s	0.000005s	0.000256s	0.000147s	0.001369s
	20.45%	17.68%	29.15%	2.92%	0.37%	18.70%	10.74%	100.00%
2.txt	0.001746s	0.000257s	0.000657s	0.000059s	0.000005s	0.000547s	0.000131s	0.003402s
	51.32%	7.55%	19.31%	1.73%	0.15%	16.08%	3.85%	100.00%
3.txt	0.005634s	0.023508s	0.156494s	0.010538s	0.000049s	0.002268s	0.093958s	0.292449s
	1.93%	8.04%	53.51%	3.60%	0.02%	0.78%	32.13%	100.00%
4.txt	0.024860s	0.183498s	1.357275s	0.056441s	0.000183s	0.005751s	0.891930s	2.519938s
	0.99%	7.28%	53.86%	2.24%	0.01%	0.23%	35.39%	100.00%
5.txt	0.086694s	1.089379s	6.747388s	0.151163s	0.000435s	0.010028s	4.854196s	12.939283s
	0.67%	8.42%	52.15%	1.17%	0.00%	0.08%	37.52%	100.00%

Tabela 2 – Tabela de tempos das operações dos algoritmos

Podemos observar que experimentos com pequenas quantidades de elementos possuem sua maior carga de tempo no momento de leitura do arquivo, devido a alocação de buffers para leitura e tempo gasto percorrendo e armazenando os pontos na estrutura de pontos [2.2.1]. Porém, fora o caso de poucos elementos, a operação que mais demora é

a de ordenar as distâncias, devido ao tamanho do vetor resultante calculado pela Equação 2.2 que aumenta quadraticamente em relação a quantidade de pontos.

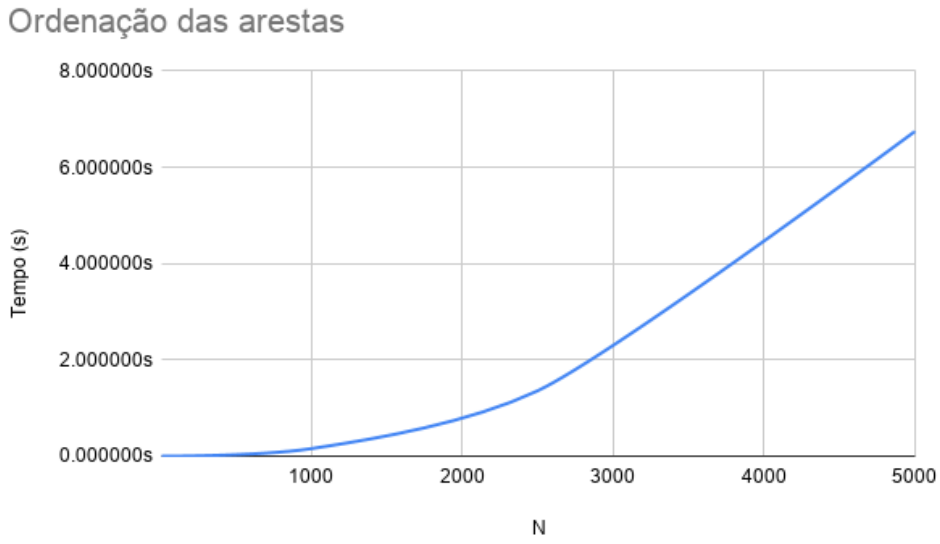

Figura 2 – Gráfico de tempo em relação a número de elementos do Cálculo das distâncias

Podemos colocar em comparação a fase do algoritmo que identifica e agrupa os pontos, que possui uma progressão linearítmica do tempo gasto, em relação a quantidade de elementos dos grupos [2.2.2].

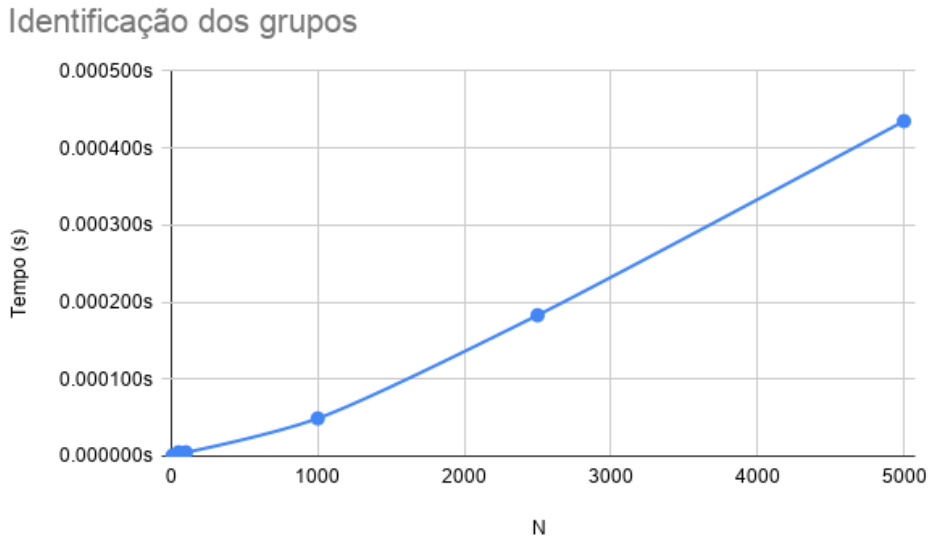

Figura 3 – Gráfico de tempo em relação a número de elementos do Cálculo das distâncias

Referências

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 9899:2011: Information technology — Programming languages — C: 7.22.5.2 The qsort function.* Geneva, CH, 2011. 355-356 p. Citado na página 5.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 9899:2011: Information technology — Programming languages — C: 7.24.4.2 The strcmp function.* Geneva, CH, 2011. 365-366 p. Citado na página 5.

KRUSKAL, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, JSTOR, v. 7, n. 1, p. 48–50, 1956. Citado na página 2.

SEDGEWICK, R. Implementing quicksort programs. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 21, n. 10, p. 847–857, 10 1978. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/359619.359631>. Citado na página 8.

SEDGEWICK, R.; WAYNE, K. *Algorithms*. 4. ed. Addison-Wesley, 2011. Acessado dia 02.03.2021. Disponível em: <https://algs4.cs.princeton.edu/15uf/>. Citado na página 5.