UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

TÉCNICAS DE BUSCA E ORDENAÇÃO

Daniel Ferrari Alves
Gabrielly Barcelos Cariman

TRABALHO PRÁTICO T1

Vitória-ES

Abril/2024

SUMÁRIO

1 INTRODUÇÃO

Neste documento, a implementação de um algoritmo de agrupamento de espalhamento máximo utilizando uma *Minimum Spanning Tree* (MST) é apresentada. O problema de agrupamento, conforme Comarela (2024), visa encontrar grupos em dados, em que objetos semelhantes estão no mesmo grupo e objetos diferentes estão em grupos distintos. Porém, existem diversos algoritmos de agrupamento disponíveis, cada um adaptado a diferentes características e contextos. A solução implementada por este trabalho, utiliza o algoritmo de *Kruskal*, visto em Wikipédia (), para obter o conjunto do agrupamento de espaço máximo.

O restante deste documento segue a estrutura delineada a seguir. No Capítulo 2, são apresentadas as principais decisões de implementação adotadas, acompanhadas de justificativas para os algoritmos e estruturas de dados selecionados. Posteriormente, no Capítulo 3, é realizada uma breve análise de complexidade sobre as principais partes da implementação. Por fim, no Capítulo 4, são apresentados os resultados da análise empírica, incluindo medições de tempo de execução para casos de teste fornecidos pelo professor e uma tabela detalhando a distribuição do tempo gasto em cada etapa do processo.

2 METODOLOGIA

2.1 Modelagem

O objetivo deste trabalho é separar os pontos em grupos distintos. Contudo, inicialmente, é imperativo que todos os pontos permaneçam unidos em um único grupo. No entanto, as arestas que os conectam devem ser sempre aquelas que possuem a menor distância entre cada par de pontos. Para resolver essa questão, será apresentada a modelagem de cada Tipo Abstrato de Dados (TAD) utilizado a seguir.

2.1.1 Ponto

A estrutura Ponto foi utilizada para representar um ponto no plano. Para isso, a estrutura Ponto possui os seguintes elementos:

- char* id: armazena a string que representa o identificador único de cada ponto.
- float* componentes: armazena o vetor de números que representam a localização do ponto no plano.
- unsigned int arv_id: armazena o identificador numérico de cada ponto.
- unsigned int arv_pid: armazena o identificador numérico da raiz de cada ponto.
- unsigned int arv_size: armazena a quantidade de pontos ligados a este ponto.

2.1.2 Distância

A estrutura Distancia representa a distância entre dois pontos. Para isso, a estrutura Distancia possui os seguintes elementos:

- float dist: armazena a distância entre o ponto de origem p1 e o ponto de destino p2.
- ponto p1: representa o ponto de origem da distância.
- ponto p2: representa o ponto de destino para o cálculo da distância.

- char connected: indica se essa distância é uma conexão entre p1 e p2, utilizado como um "boolean"

2.1.3 Lista

A estrutura Lista é utilizada para armazenar todos os pontos no formato de lista no momento em que os pontos são lidos do arquivo e inicializados na memória do programa. A escolha dessa estrutura foi devido ao fato de que utiliza alocação dinâmica de memória, diferentemente de array, que teria de saber a quantidade total de pontos. Como não é possível saber a quantidade total de pontos sem ler o arquivo inteiro até o fim, optou-se por criar e armazenar os pontos em uma lista simplesmente encadeada com sentinela, conforme é lido linha por linha do arquivo.

- celula prim: armazena o ponteiro para a primeira célula da lista.
- celula ult: armazena o ponteiro para a última célula da lista.
- unsigned int size: armazena o tamanho total da lista, isto é, a sua quantidade de elementos.
- unsigned int qtdEixos: armazena a quantidade de coordenadas que um ponto tem.

A escolha de utilizar uma lista com sentinela foi feita para armazenar seu tamanho e a quantidade de coordenadas de um ponto.

2.1.4 Array

A estrutura Array foi utilizada para armazenar dois tipos de arrays diferentes: o array de pontos e o array com todas as distâncias.

Essa estrutura foi escolhida devido à sua capacidade de realizar ordenações de forma mais eficiente do que comparado com a lista encadeada.

Por isso, após a leitura completa da entrada e com os pontos armazenados na lista, foi criado o array para os pontos e eles foram armazenados nesse array.

- void** array: armazena o vetor de tipo genérico fornecido.

- unsigned int qtdInseridos: armazena a quantidade total de elementos inseridos no array.
- unsigned int qtdEixos: armazena a quantidade de coordenadas que um ponto tem.

2.2 Funções Importantes

Além dos arquivos .h e .c de cada TAD apresentado, existem também os arquivos tree.h e tree.c, responsáveis por toda a lógica necessária para a construção da MST. A manipulação dos arquivos de entrada e saída é realizada nos arquivos arquivo.h e arquivo.c, a lógica principal do programa é feita nos arquivos tree.c e tree.h e o restante servem para suprir esses arquivos, com destaque na função para ordenar um array, que é uma parte crítica do programa em termos de otimização.

2.2.1 Sort do Array

Foi optado por usar o Quick Sort (BRUNET, 2019a), por ser in-place, economizando memória, e o caso médio e melhor caso terem ordem de complexidade de $O(n * log(n))$. Tendo em vista que o pior caso é quando não tem nenhum elemento de um lado do pivot e todos os outros do outro lado, foi sempre escolhido como pivot a mediana entre o primeiro elemento, o elemento do meio e o último elemento, evitando assim o pior caso (com ordem de complexidade $O(n$š). Foram implementadas e testadas duas variantes do Quick Sort, uma utilizando o particionamento de Lomuto e outra utilizando o particionamento de Hoare (BRUNET, 2019b). A escolha entre as 2 variantes será discutida no Capítulo 4

2.2.2 Tree

O arquivo tree contém as funções que estão envolvidas na construção da Minimum Spanning Tree (MST) e na ordenação dos pontos de acordo com os grupos para a saída do arquivo.

2.2.2.1 void uf_union(ponto p, ponto q, array pts)

A função uf_union recebe dois pontos p e q e um array pts que contém todos os pontos do conjunto. Foi criada uma variante inspirada na estrutura Weighted quick-union (COMA-

RELA,), considerando as particularidades do projeto como ter que desconectar 2 pontos e a ordem dos elementos na árvore importarem. Ela encontra as raízes dos pontos p e q, compara o tamanho das árvores representadas por esses pontos e, em seguida, os une de acordo com o tamanho das árvores. Se a árvore de p for menor que a de q, os conjuntos são unidos e p é pendurado sob q; caso contrário, q é pendurado sob p.

2.2.2.2 unsigned int uf_find(ponto p, array pts)

Nesse caso, o find pôde ser uma aplicação direta do find da estrutura Weighted quick-union (COMARELA,), pelo fato das particularidades do projeto não interferirem no find. Não foi possível implementar path halving, uma vez que a ordem dos elementos das árvores importam para o resultado final.

2.2.2.3 void uf_disconnect(distancia dist, array pts)

O código implementa a função uf_disconnect, que é responsável por desconectar dois pontos em uma estrutura de dados de conjuntos disjuntos. A função verifica qual ponto é filho do outro e faz com que ele se torne a raiz (se tornando seu próprio pai), além de reduzir o tamanho da árvore do ponto pai. Isso é feito para manter a estrutura balanceada nas futuras operações de union.

2.2.2.4 void kruskal_getMST(array dists, array pts)

A função kruskal_getMST gera uma Minimum Spanning Tree (MST) a partir de um array de distâncias entre pontos e um array de pontos. Ele percorre o array ordenado de distâncias e conecta os pontos, formando a árvore, desde que eles não estejam conectados previamente.

2.2.2.5 void tree_make(array dists, array pts, unsigned int qtdToRemove)

A primeira função que é externalizada (está no .h). Ela chama a função kruskal_getMST e depois remove as maiores arestas de acordo com qtdToRemove, gerando os grupos separados.

2.2.2.6 void tree_sort(array pts, unsigned int qtdGrupos)

A função tree_sort organiza os pontos em grupos e os ordena para facilitar a saída no arquivo. Isso é feito em passos que incluem:

1. Criar um array auxiliar para ordenação.
2. Contar a quantidade de elementos em cada grupo e atualizar essa informação nos pontos.
3. Criar um array grupos contendo as raízes de cada grupo.
4. Criar sub-arrays para cada grupo, ordená-los internamente e inseri-los no array auxiliar.
5. Apagar e criar um novo array grupos, contendo o primeiro elemento de cada grupo.
6. Ordenar o array grupos com a ordem alfabética.
7. Reorganizar os grupos no array auxiliar de acordo com a ordem do array grupos
8. Copiar os elementos ordenados do array auxiliar de volta para o array principal.
9. Liberar a memória alocada.

3 ANÁLISE DE COMPLEXIDADE

Neste capítulo, será feita uma análise de complexidade sobre as principais partes da implementação desenvolvida, buscando identificar os pontos críticos e compreender como o algoritmo se comporta sob diferentes condições

3.1 Array Sort

O estudo sobre o array sort se deve ao fato de que a ordenação do array de distâncias é um ponto crítico, visto que é a maior estrutura de dados utilizada no programa. O método de ordenação de arrays utilizado foi o Quick Sort que, conforme discutido no Capítulo 2, possui uma complexidade teórica de $O(n*log(n))$ no melhor caso e no caso médio, e complexidade teórica de $O(n^2)$ no pior caso, sendo que esse é evitado na implementação. Para definir a complexidade do Quick Sort, primeiro decidir qual parâmetro será analisado (nesse caso, irá ser considerado a quantidade de vezes que a função comparator foi chamada), depois deve-se definir a complexidade da função de partição, da função de escolher o pivot e depois definir a complexidade da função Quick Sort.

3.1.1 Partição de Lomuto

Figura 1 – Partição de Lomuto

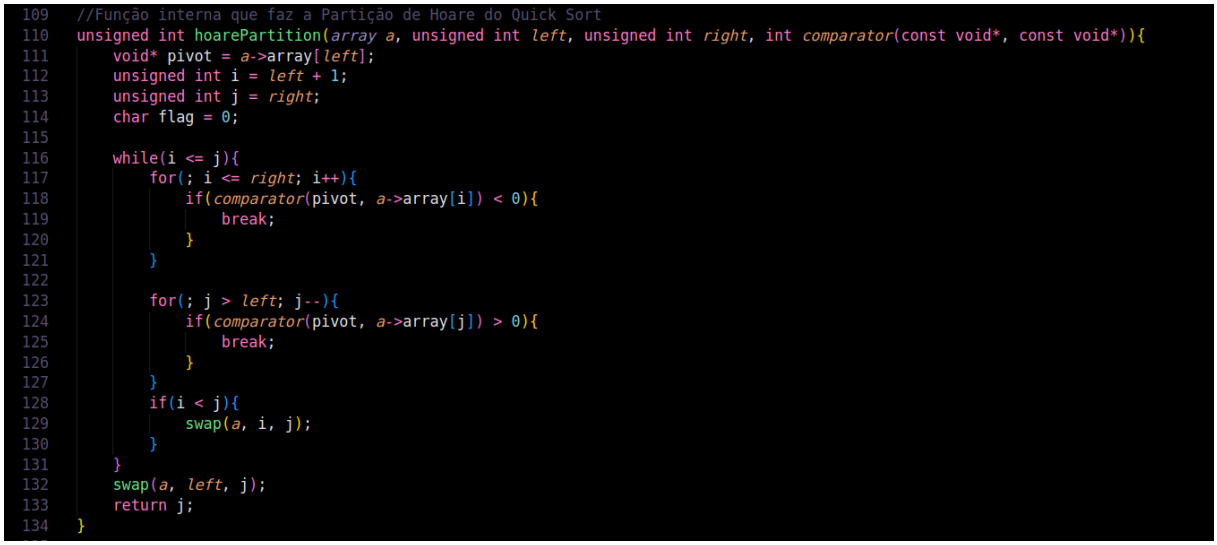
```
94  //Função interna que faz a Partição de Lomuto do Quick Sort
95  unsigned int lomutoPartition(array a, unsigned int left, unsigned int right, int comparator(const void*, const void*)){
96      void* pivot = a->array[left];
97      unsigned int i = left;
98
99      for(unsigned int j = left+1; j <= right; j++){
100         if(comparator(pivot, a->array[j]) > 0){
101             i++;
102             swap(a, i, j);
103         }
104     }
105     swap(a, left, i);
106     return i;
107 }
```

Fonte: Produção do próprio autor.

Na função que executa a Partição de Lomuto, a função comparator é chamada uma vez dentro do for. Em todos os casos, a função comparator será chamada $N - 1$ vezes (lembrando que esse N é o tamanho do array passado para a função de partição), tendo uma complexidade de $C(N - 1)$, $\sim (N)$ ou $O(N)$.

3.1.2 Partição de Hoare

Figura 2 – Partição de Hoare



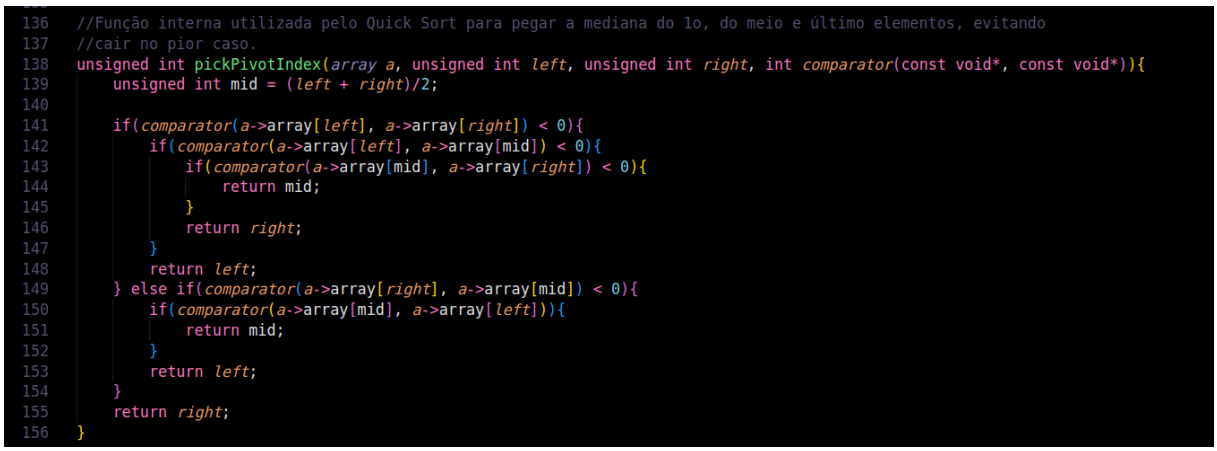
```
109  //Função interna que faz a Partição de Hoare do Quick Sort
110  unsigned int hoarePartition(array a, unsigned int left, unsigned int right, int comparator(const void*, const void*)){
111      void* pivot = a->array[left];
112      unsigned int i = left + 1;
113      unsigned int j = right;
114      char flag = 0;
115
116      while(i <= j){
117          for(; i <= right; i++){
118              if(comparator(pivot, a->array[i]) < 0){
119                  break;
120              }
121          }
122
123          for(; j > left; j--){
124              if(comparator(pivot, a->array[j]) > 0){
125                  break;
126              }
127          }
128          if(i < j){
129              swap(a, i, j);
130          }
131      }
132      swap(a, left, j);
133      return j;
134  }
```

Fonte: Produção do próprio autor.

Já na função que executa a Partição de Hoare, a função comparator é chamada uma vez dentro do for do i e uma vez dentro do for do j. Porém, como quando os indicadores i e j se cruzam a função sai do loop, acaba que cada uma dessas chamadas são chamadas $(N-1)/2$ vezes (lembrando que N é o tamanho do array passado para a função). Logo, a função tem uma complexidade de $C(N-1)$, $\sim (N)$ ou $O(N)$.

3.1.3 pickPivotIndex()

Figura 3 – pickPivotIndex()



```
136  //Função interna utilizada pelo Quick Sort para pegar a mediana do 1o, do meio e último elementos, evitando
137  //cair no pior caso.
138  unsigned int pickPivotIndex(array a, unsigned int left, unsigned int right, int comparator(const void*, const void*)){
139      unsigned int mid = (left + right)/2;
140
141      if(comparator(a->array[left], a->array[right]) < 0){
142          if(comparator(a->array[left], a->array[mid]) < 0){
143              if(comparator(a->array[mid], a->array[right]) < 0){
144                  return mid;
145              }
146              return right;
147          }
148          return left;
149      } else if(comparator(a->array[right], a->array[mid]) < 0){
150          if(comparator(a->array[mid], a->array[left])){
151              return mid;
152          }
153          return left;
154      }
155      return right;
156  }
```

Fonte: Produção do próprio autor.

A análise da função pickPivotIndex() é mais simples, uma vez que ela chama a função comparator uma quantidade fixa de vezes. No melhor caso temos uma complexidade de $C(2)$ e no pior caso $C(3)$. Em ambos o caso a complexidade é de $\sim (0)$ ou $O(0)$, tornando-a irrelevante.

3.1.4 quickSort()

Figura 4 – quickSort()

```
158  //Função interna com implementação do Quick Sort para a struct Array.
159  //O índice do pivot é escolhido usando a media do 1o, do meio e últimos elementos, para evitar
160  //cair no pior caso.
161  void quickSort(array a, unsigned int left, unsigned int right, int comparator(const void*, const void*),
162                 unsigned int partition(array, unsigned int, unsigned int, int comparator(const void*, const void*))){
163
164      if(left < right){
165          unsigned int pivotIndex = pickPivotIndex(a, left, right, comparator);
166          swap(a, left, pivotIndex);
167
168          pivotIndex = partition(a, left, right, comparator);
169          if(pivotIndex > 0){
170              quickSort(a, left, pivotIndex-1, comparator, partition);
171          } else{
172              quickSort(a, left, 0, comparator, partition);
173          }
174          quickSort(a, pivotIndex+1, right, comparator, partition);
175      }
176
177  }
```

Fonte: Produção do próprio autor.

A função quickSort() vai pegar o array e dividir ele em 2: a parte à esquerda do pivot (menores que o pivot) e a parte à direita do pivot (maiores que o pivot), e vai se chamar recursivamente para cada uma dessas partes. Para posicionar o pivot, inicialmente chama a função de partição (antes de dividir o array), cuja complexidade é de $\sim (N)$ ou $O(N)$. O melhor caso seria quando as duas partes fossem de tamanhos iguais, de forma que se teria:

$$C(N) = C(N-1) + C((N-1)/2) + C((N-1)/2)$$

$$C(1) = C(1-1) + C(0/2) + C(0/2) = 0, N = 1$$

$$C(2) = C(2-1) + C(1/2) + C(1/2) = 2, N = 2$$

$$C(4) = C(4-1) + C(3/2) + C(3/2) = 6, N = 4$$

$$C(8) = C(8-1) + C(7/2) + C(7/2) = 14, N = 8$$

$$C(16) = C(15-1) + C(15/2) + C(15/2) = 30, N = 16$$

$$C(N) =$$

Obs.: Não se conseguiu concluir a parte de análise de complexidade do relatório antes do prazo de entrega.

4 ANÁLISE EMPÍRICA

Neste capítulo, iremos realizar uma análise empírica do desempenho do algoritmo implementado. Para isso, foram registrados os tempos de execução de diferentes etapas, incluindo leitura e escrita de arquivos. Utilizamos a biblioteca time.h com a função clock() para medir o tempo de início e fim de cada operação. Iniciamos com a leitura dos dados, seguida pela cópia para um array e o cálculo das distâncias entre pontos. Depois, registramos o tempo de ordenação das distâncias e a obtenção da Árvore Geradora Mínima (MST), além da identificação dos grupos e escrita do arquivo de saída. Os tempos medidos foram usados para construir uma tabela detalhando a distribuição percentual do tempo gasto em cada etapa.

4.1 Resultados

Na Tabela 5, está representado o tempo total de execução.

Tabela 1 – Tempo total

Arquivo	n	k	m	Tempo (s)
1	50	2	2	0.000523
2	100	4	3	0.002223
3	1000	5	2	0.644296
4	2500	5	5	4.000382
5	5000	10	19	19.661541

Em que n é o número de linhas do arquivo de entrada, k representa o número de conjuntos a serem formados e m é a dimensão dos pontos

Depois disso, foi feita a medição do tempo das seguintes etapas:

Tabela 2 – Tempo Leitura

Arquivo	Tempo Total (s)	Tempo Leitura (s)	Tempo Leitura (%)
1	0.000523	0.000060	11.47
2	0.002223	0.000109	4.90
3	0.644296	0.001790	0.27
4	4.000382	0.013600	0.34
5	19.661541	0.049805	0.25

Tabela 3 – Tempo Cálculo Distância

Arquivo	Tempo Total (s)	Tempo Cálculo (s)	Tempo Cálculo (%)
1	0.000523	0.000100	19.12
2	0.002223	0.000320	14.39
3	0.644296	0.030994	4.81
4	4.000382	0.186212	4.65
5	19.661541	0.960098	4.88

Tabela 4 – Tempo Ordenação

Arquivo	Tempo Total (s)	Tempo Ordenação (s)	Tempo Ordenação (%)
1	0.000523	0.000166	31.73
2	0.002223	0.000814	36.61
3	0.644296	0.157768	24.48
4	4.000382	1.29833	32.45
5	19.661541	7.324673	37.25

Tabela 5 – Tempo Fracionado

Arquivo Grupos	Leitura Escrita	Cálculo Distância	Ordenação Distâncias	MST
1	0.000060 (11,47%)	0.000049 (9,37%)	0.000166 (31,74%)	0.000112 (21,41
0.000024 (4,59%)	0.000099			
2	0.000109 (0,49%)	0.000354 (1,59%)	0.000814 (3,66%)	0.000641 (2,88%
0.000075	0.000184			
3	0.001790	0.023960	0.157768	0.402769
0.038624	0.000166			
4	0.013600	0.179070	1.298333	1.913142
0.436499	0.000453			
5	0.049805	0.970634	7.324673	8.343234
2.142321	0.000893			

REFERÊNCIAS

BRUNET, J. A. Ordenação por Comparação: Quick Sort. 2019. Disponível em: <https://joaoarthurbm.github.io/eda/posts/quick-sort/>. Citado na página 5.

BRUNET, J. A. Particionamento Hoare para o Quick Sort. 2019. Disponível em: <https://joaoarthurbm.github.io/eda/posts/particionamento-hoare/>. Citado na página 5.

COMARELA, G. V. Conectividade Dinâmica (parte 2). Disponível em: <https://ava.ufes.br/pluginfile.php/668081/mod_resource/content/4/slides.pdf>. Acesso em: 17 abril 2024. Citado na página 6.

COMARELA, G. V. Trabalho Prático T1. 2024. Disponível em: <https://ava.ufes.br/pluginfile.php/668056/mod_resource/content/6/spec.pdf>. Acesso em: 17 abril 2024. Citado na página 2.

WIKIPÉDIA. Kruskal's algorithm. Disponível em: <https://en.wikipedia.org/wiki/Kruskal%27s_algorithm>. Acesso em: 17 abril 2024. Citado na página 2.