1. INTRODUÇÃO

O Agrupamento de Espaçamento Máximo é uma tarefa interessante em muitas áreas, desde a análise de dados até a otimização de rotas. Trata-se de um conceito abstrato porém simples de ser implementado computacionalmente. Um dos métodos mais comuns de agrupamento envolve a divisão de um conjunto de elementos em k grupos, de tal forma que a distância entre os elementos em grupos diferentes seja maximizada, e que a distância entre elementos do mesmo grupo seja minimizada. Uma abordagem popular para resolver esse problema é usar o algoritmo de Kruskal, que é comumente utilizado para encontrar a árvore geradora mínima (MST) de um grafo ponderado. A seguir será apresentada uma implementação específica do algoritmo de Kruskal para resolver os diversos casos de problemas de agrupamento de espaçamento máximo. Além disso, serão discutidas as decisões e métodos utilizados para o desenvolvimento do programa, incluindo a escolha de estruturas de dados e algoritmos auxiliares. Além disso, será apresentada uma análise da complexidade de cada parte do programa e uma medida experimental dos tempos de execução.

2. METODOLOGIA

De maneira geral o algoritmo completo se divide em quatro partes fundamentais, sendo essas a leitura do arquivo de entrada, o cálculo de todos os pesos do total de arestas possíveis, a ordenação destas arestas, a geração da MST e eliminação das maiores arestas e por fim a geração dos grupos conexos. Inicialmente o programa faz a coleta de informações úteis na manipulação posterior dos dados como o número de pontos contidos no arquivo de entrada, o número de coordenadas do espaço utilizado para a entrada e o número de componentes que deverão estar presentes na saída. Em seguida os dados coletados são usados para a leitura das linhas do arquivo que são logo devidamente ordenadas em ordem lexicográfica, sendo a partir dessa nova ordenação que os dados são coletados e armazenados de forma simples em estruturas estáticas por facilidade de implementação. Como as linhas do arquivo já foram alocadas dinamicamente, aproveitamos

esse espaço de memória para armazenar os identificadores dos pontos apenas incluindo o caractere finalizador de strings na própria linha. O algoritmo segue com o cálculo dos pesos das arestas utilizando dois loops do tipo for, um método mais usual de verificação de pares. Os pesos foram calculados usando o quadrado das distâncias para evitar o uso de algoritmos matemáticos e os dados foram armazenados em uma estrutura chamada “edg” que armazena os índices dos seus respectivos pontos com relação aos vetores previamente alocados de forma automática. Utilizando a função da biblioteca padrão que implementa o quicksort conseguimos todas as arestas ordenadas para serem usados dentro do algoritmo de Kruskal responsável pela geração da Minimum Spanning Tree (MST). Neste, são necessárias a verificação de uma parcela dos pesos e arestas visto que estes estão ordenados e usamos apenas parte (N-K) do total de arestas da árvore (N-1), dessa forma dando a característica de clustering para o programa. Para a implementação do algoritmo gerador da MST foi utilizada a estrutura típica de um problema union-find chamada “set” usando especificamente uma implementação usando árvores sendo esta aprimorada para a seleção de pesos para tais arvores, uma solução conhecida como “Weighted Quick-Union” escolhida devido a complexidade logarítmica das operações de união e teste de conexão. Após a construção de todas as componentes, as listas de pontos precisam ser apresentadas em ordem lexicográfica tanto para a relação entre os pontos dentro dos grupos como para os grupos em si. Para tal são identificadas todas as componentes em comum entre os pontos usando a função “find” da estrutura “set” (a complexidade de cada fase do programa será discutida posteriormente) e em seguida devido a ordenação das linhas no início do programa, uma das propriedades dessa parte do algoritmo e que a saída fica naturalmente ordenada exatamente da maneira como foi proposto, pois as entradas são escolhidas sequencialmente de menor para maior. Após a construção do arquivo de saída o programa pode prosseguir em suas devidas finalizações.

3. ANÁLISE DE COMPLEXIDADE

O primeiro processo realizado no programa será a contagem do número de pontos a serem adicionados e a quantidade de dimensões usadas na entrada do programa. Assumindo que o número de pontos será uma característica muito mais influente nas diversas fases do programa, podemos supor a contagem de coordenadas como uma operação de complexidade constante, ou seja, O(1). Para a contagem de pontos usamos a contagem de linhas no arquivo de entrada como fonte dessa informação, sendo desta forma um processo de complexidade linear denotada por O(N). Tais operações têm apenas caráter de coleta de informações básicas requerendo que o arquivo de entrada seja reiniciado para uma segunda leitura que dessa vez terá o caráter de leitura das informações. Sendo assim uma fase de complexidade O(N) adicional na execução do programa. Após a coleta de dados, os pesos das arestas serão calculados. Devido a suposição da entrada da quantidade de coordenadas ser considerada constante o cálculo de uma distância tem complexidade também constante. O processo de coleta de todas as distâncias têm uma complexidade mínima visto que são necessários obrigatoriamente o cálculo de N(N-1)/2 distâncias deixando essa fase do programa com complexidade O(N^2) sendo essa a parte mais ineficiente do programa inteiro. A construção da MST envolve um processo não tão bem comportado como o cálculo das distâncias e apresenta uma grande influência estatística visto que este depende da disposição dos pontos. No entanto, é possível aproximar a complexidade ao ignorar o efeito de conexões que não unem duas novas componentes. Sabendo que uma árvore de N pontos sem ciclos tem exatamente N-1 arestas e sabendo que não estamos considerando as K-1 maiores arestas temos (N-1)-(K-1) = N-K arestas sendo consideradas. Além disso, assumindo o uso da função union de complexidade O(logN) para cada conexão temos uma complexidade de (N-K)*logN que pode ser representada por O(NlogN). E por fim o processo de apresentação das saídas envolve uma sequência do uso de funções “find” para a identificação única de cada componente pela raiz da árvore de cada componente criando uma estrutura muito parecida com a usada no uso da estrutura Quick-Find. Assumindo N pontos, temos que a função “find” de complexidade logN faz com que essa fase do código tenha complexidade NlogN. Em seguida percorremos o array uma sequência de vezes para a

apresentação de cada componente na ordem correta (assumindo que as chaves dos pontos já foram devidamente ordenadas). Dessa forma como temos K componentes e precisamos percorrer o array para apresentar cada componente temos uma complexidade de K*N que pode ser representada na forma de O(N). Por fim os processos de liberação de memória que foram eventualmente necessários nessa última parte podem ser aproximados por O(N) já que a maior parte dessas alocações envolvem características derivadas do número total de pontos fornecidos na entrada.

4. ANÁLISE EMPÍRICA

Para a análise empírica usamos uma bateria de 5 entradas variando o número de pontos de 50 a 5000. Foram calculados os tempos médios de execução de cada parte do algoritmo (feitos em uma máquina com capacidade de processamento de 4GB). Foram verificados os seguintes resultados:

Entradas	Leitura da Entrada (s)	Cálculo das Distâncias (s)	Ordenação das Distâncias (s)	Formação das componentes (s)	Tempo total
50	0.000136	0.000039	0.000155	0.000020	0.000461
100	0.000338	0.000168	0.001031	0.000066	0.001812
1000	0.001494	0.013227	0.096891	0.001700	0.114143
2500	0.005882	0.112269	0.718436	0.082932	0.923641
5000	0.019376	0.687470	3.303062	0.491730	4.513880

Considerando os conceitos fundamentais de ordem de crescimento da complexidade de tempo dos algoritmos. Os resultados experimentais de fato seguem o modelo considerando as variações no tamanho da entrada e as diferenças de tempo entre as diversas partes do código para entradas maiores.