UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Isabela Gratts dos Santos
Gianluca Schmidt Mantovaneli
Renan Campista

Agrupamento de espaçamento máximo

VITÓRIA, ES
2024

Sumário

1- Introdução

O propósito deste trabalho é entender o funcionamento e implementar um algoritmo de espaçamento máximo. Para tal, utilizamos o algoritmo de Kruskal a fim de encontrar uma árvore geradora mínima.

A ideia básica por trás do algoritmo de Kruskal é selecionar arestas do grafo em ordem crescente de peso e adicioná-las à árvore geradora mínima, garantindo que não forme ciclos fechados. Isso é feito até que todos os vértices estejam conectados na árvore.

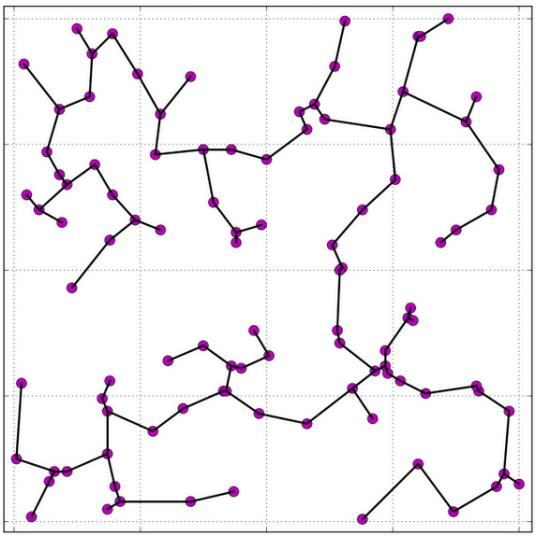

Figura 1: Representação de uma árvore geradora mínima

Também foi utilizada uma estrutura union-find (estrutura de dados de conjuntos disjuntos), que nos forneceu operações necessárias para encontrar conjuntos de pontos, unir dois conjuntos, e verificar se dois pontos estão no mesmo conjunto de forma eficiente.

2- Metodologia

2.1- Armazenamento dos Pontos

A estrutura `Point` é responsável por armazenar um ponto no espaço m-dimensional. Ela consiste em um array de caracteres para o identificador (`char *id`) e outro array de ponto flutuante para as coordenadas do ponto (`double *coordinate`). Todos os pontos lidos na entrada são salvos na estrutura `CartesianPlane`, que contém um vetor de ponteiros para `Point`, juntamente com um inteiro que representa a dimensão do espaço. A vantagem de armazenar a dimensão do espaço em uma variável é evitar o uso frequente da função `realloc` durante a leitura das coordenadas dos pontos, já que a mesma é considerada uma operação computacional custosa. Portanto, basta alocar dinamicamente a memória uma única vez no construtor da estrutura `Point`.

2.2- Estratégia de Leitura dos Pontos

Antes de iniciar a leitura dos pontos, é essencial determinar a dimensão do espaço. Isso evita a necessidade de redimensionar a memória alocada, como mencionado anteriormente. Uma vez que os dados de um ponto são separados por vírgulas, a contagem do número delas em uma linha do arquivo de entrada nos dá a dimensão da coordenada. Após obter a dimensão do espaço, o programa retorna ao início do arquivo utilizando a função `fseek` para começar a leitura dos pontos.

2.3- Sequência do Algoritmo de Kruskal

Após a leitura de todos os pontos, a função `kruskal_solve` é invocada para executar o algoritmo de Kruskal, sendo ela a principal função, que busca encontrar uma árvore geradora mínima. Antes de iniciar o algoritmo, é realizada a ordenação dos pontos de acordo com o seu identificador, uma vez que a especificação do trabalho solicita que na saída os elementos estejam em ordem lexicográfica. Para executar o algoritmo, é utilizada a estrutura `Kruskal`, que armazena um ponteiro para a estrutura `CartesianPlane`, a qual aponta para o plano cartesiano lido. Além disso, a estrutura `Edge` é de extrema importância, pois permite visualizar a distância entre dois pontos como uma aresta. Cada `Edge` contém dois inteiros que armazenam os índices de pontos conectados e um ponto flutuante para representar o peso dessa aresta, que no caso é a distância euclidiana entre os pontos.

A escolha da estrutura *Edge* em vez de uma matriz para armazenar as distâncias euclidianas é justificada pelo fato de que apenas uma diagonal da matriz é necessária para salvar todas as distâncias. Utilizando um array da estrutura *Edge*, não há desperdício de memória, uma vez que é possível alocar o array com tamanho igual o número total de arestas, dado pela fórmula n*(n-1)/2, sendo “n” o número total de pontos. Além disso, o array simplifica o processo de ordenação das arestas, que é uma etapa do algoritmo.

A próxima etapa do algoritmo é inicializar os arrays *parent* e *sz*, presentes na estrutura *Kruskal*, que serão usados para implementar a estrutura de dados de conjuntos disjuntos. Isso é realizado na função *populate_edges_and_parents*, que também calcula o peso de cada aresta, a partir do cálculo da distância euclidiana. É importante mencionar que para o cálculo da distância euclidiana, não é retornada a raiz quadrada da soma dos quadrados, e sim apenas a soma. O motivo disso é que a raiz quadrada é uma operação computacional cara e não é necessária para comparação, visto que se d1 < d2, então sqrt(d1) < sqrt(d2).

A próxima etapa do algoritmo é processar as arestas para encontrar a árvore geradora mínima. Na função *process_edges*, o loop iniciado continua até que o número de arestas seja menor que o número de pontos no plano cartesiano menos o número de grupos. Isso garante que todas as arestas necessárias para conectar todos os pontos sejam consideradas. Dentro do loop, é verificado se os pontos nas extremidades da aresta atual estão no mesmo conjunto. Se estiverem, a aresta é ignorada, pois adicioná-la criaria um ciclo. Se os pontos não estiverem no mesmo conjunto, eles são unidos em um único conjunto com o uso da estrutura de dados de conjuntos disjuntos, que é eficiente com operações de união e busca. Isso representa a adição da aresta à árvore geradora mínima.

2.4- Impressão dos conjuntos

Por fim, é realizada a impressão dos grupos por meio da função *kruskal_print_groups*. Para isso, é utilizado um array de booleanos para garantir que um mesmo ponto seja impresso uma única vez. Como dito anteriormente, os pontos foram ordenados antes de iniciar o algoritmo. Dessa forma, é possível encontrar o primeiro representante (por ordem lexicográfica) do conjunto. Em seguida, basta iterar sobre o restante do array e verificar se há mais pontos pertencentes ao grupo atual. Dessa forma, a identificação dos grupos ocorre no momento da impressão, visto que na etapa anterior é obtida apenas a árvore geradora mínima.

3- Bug conhecido

Durante os testes realizados no código desenvolvido, identificamos um pequeno problema que ocorria esporadicamente ao executar o programa com a ferramenta Valgrind. O erro "Process terminating with default action of signal 27 (SIGPROF)" indica que o programa foi encerrado devido ao tempo limite de profiling (perfilamento) ter sido excedido. Geralmente, isso ocorre quando o Valgrind está tentando realizar o perfilamento do programa, mas ultrapassa o tempo limite definido. É importante ressaltar que esse tipo de erro não está diretamente relacionado ao código do programa, mas sim ao tempo necessário para que o Valgrind conclua o profiling completo. É relevante mencionar também que esse erro não afeta a saída do programa.

```
==1110==
==1110== Process terminating with default action of signal 27 (SIGPROF)
==1110==    at 0x497BA1A: __open_nocancel (open64_nocancel.c:39)
==1110==    by 0x498A56F: write_gmon (gmon.c:370)
==1110==    by 0x498ADDE: _mcleanup (gmon.c:444)
==1110==    by 0x48A7494: __run_exit_handlers (exit.c:113)
==1110==    by 0x48A760F: exit (exit.c:143)
==1110==    by 0x488BD96: (below main) (libc_start_call_main.h:74)
==1110==
==1110== HEAP SUMMARY:
==1110==     in use at exit: 16,600 bytes in 1 blocks
==1110==   total heap usage: 43 allocs, 42 frees, 27,854 bytes allocated
==1110==
==1110== LEAK SUMMARY:
==1110==    definitely lost: 0 bytes in 0 blocks
==1110==    indirectly lost: 0 bytes in 0 blocks
==1110==      possibly lost: 0 bytes in 0 blocks
==1110==    still reachable: 16,600 bytes in 1 blocks
==1110==         suppressed: 0 bytes in 0 blocks
==1110== Reachable blocks (those to which a pointer was found) are not shown.
==1110== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==1110==
==1110== For lists of detected and suppressed errors, rerun with: -s
==1110== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
make: *** [Makefile:16: valgrind] Profiling timer expired
```

Figura 2: Erro do Valgrind "Process terminating with default action of signal 27 (SIGPROF)"

4- Análise de complexidade

Para analisar a complexidade do nosso código selecionamos as seguintes funções para analisar: leitura dos dados, cálculo das distâncias, ordenação das distâncias, obtenção da árvore geradora mínima, identificação dos grupos, escrita do arquivo de saída, elas foram escolhidas por serem as funções mais relevantes dentro do código, para que se tenha uma boa leitura da complexidade do código como um todo.

4.1- Leitura dos dados

Esta função possui um loop, que percorre cada linha do arquivo. Cada linha do arquivo corresponde a um ponto no plano, logo pode-se assumir que esse loop sempre vai ocorrer N vezes, sendo N a quantidade de pontos.Importante lembrar que em cada linha o código percorre cada coordenada, ou seja, M vezes, sendo M a dimensão do plano cartesiano, sendo assim temos uma complexidade de aproximadamente $O(N * M)$.

4.2- Cálculo das distâncias

Para analisar esta função temos que olhar para tempo usado para calcular apenas 1 ponto, como cada entrada pode ter pontos em um espaço com diferentes dimensões, o cálculo de um ponto sempre será um loop que será percorrido M vezes, sendo M a dimensão do plano cartesiano. Porém, este cálculo acontece dentro de um loop duplo, que por sua vez tem uma complexidade de $O\ (\frac{N^2-N}{2})$, onde N representa a quantidade de pontos. Portanto, pode-se dizer que no fim temos uma complexidade de $O\ (\frac{N^2-N}{2} * M)$.

4.3- Ordenação das distâncias

Nesta parte do programa utilizamos um quicksort que é nativo dentro das bibliotecas de C. Como um algoritmo já conhecido, ele tem uma complexidade de $O(N\ logN)$ no melhor dos casos ou caso médio. No pior caso pode chegar a $O(N^2)$. Porém, isso tem menos chance de acontecer quando o N cresce muito.

4.4- Obtenção da MST

Neste código, a obtenção da árvore mínima acontece dentro da função *process_edges*, dentro dessa função tem um loop que continua enquanto o número de arestas processadas *num_edges* for menor que o número total de pontos N

menos k, que é o número de grupos passado na entrada. É importante mencionar que dentro do loop, são utilizadas as funções *UF_connected* e *UF_union*, que possuem complexidades O(α(n)), onde α(n) é a função inversa do crescimento de Ackermann, que é praticamente constante na prática para tamanhos razoáveis de n. Portanto, a obtenção da MST tem uma complexidade de aproximadamente $O(N - k)$.

4.5- Identificação dos grupos

Para resolver este problema nós resolvemos não salvar os grupos em novas estruturas. Os grupos são “identificados” na hora de imprimir, percorrendo as árvores verificando as arestas que já foram impressas, se o ponto já foi visitado o programa não entra no segundo for, e como está tudo ordenado, o resultado final são os grupos printados cada um na sua linha, e economizamos processamento. No fim das contas podemos afirmar que a complexidade dessa parte é de $O(N)$ sendo N o número total de pontos. Assim como na obtenção da árvore mínima, é desconsiderada a complexidade da função *UF_find*, pois como foi dito anteriormente, na prática sua complexidade é praticamente constante.

4.6- Escrita do arquivo de saída

Como já foi descrito antes os pontos são impressos paralelamente a identificação dos grupos, mais à frente no relatório será demonstrado o quanto esse ganho de desempenho deixou o código ágil no fim das contas.

5- Análise Empírica

Depois de realizar alguns testes no programa, temos os seguintes resultados:

Arquivo de entrada	Número de elementos (n)	K	Quantidade de Coordenadas (m)	Tempo médio total de execução (Segundos)
0.txt	10	3	2	0.000000
1.txt	50	2	2	0.000000
2.txt	100	4	3	0.000000
3.txt	1000	5	2	0.260250
4.txt	2500	5	5	1.794934
5.txt	5000	10	10	7.870868

Tabela 1 : Tempo médio total gasto em cada teste.

Com a tabela 1 percebemos que há uma relação exponencial entre a quantidade de pontos analisados e o tempo gasto pelo programa, quantos mais pontos ele processa mais tempo ele leva. Entretanto, não se pode esquecer que a quantidade de dimensões pode impactar um pouco na velocidade. Porém, isso se mostra mais relevante apenas no cálculo das distâncias das arestas, pois quanto mais coordenadas, mais cálculos vão ocorrer em cada ponto.

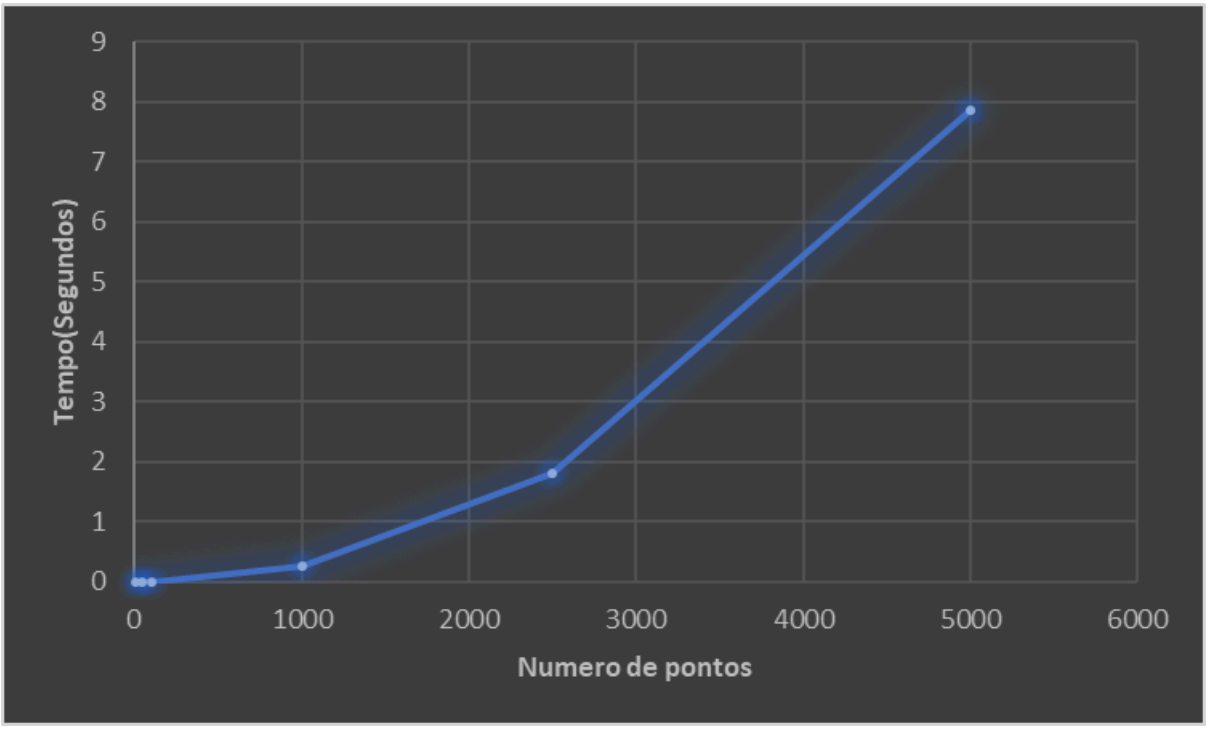

Figura 3: Gráfico da relação entre tempo gasto e número de pontos.

O gráfico apresentado na figura 3 evidencia essa relação, o maior teste é de 5000, porém já dá uma ideia da tendência que o gráfico vai seguir. é uma exponencial, onde quanto mais pontos são entregues maior será o tempo gasto no total pelo código.

Nome do teste	Leitura de dados	Cálculo das distâncias	Ordenação das distâncias	Obtenção da MST	Identificação dos grupos	Escrita do Arquivo de Saída
0.txt	-	-	-	-	-	-
1.txt	-	-	-	-	-	-
2.txt	-	-	-	-	-	-
3.txt	0.14 %	26.29 %	73.58 %	0.00 %	0.00 %	0.00 %
4.txt	0.00 %	32.44 %	67.56 %	0.00 %	0.00 %	0.00 %
5.txt	0.17 %	32.01 %	67.51 %	0.31 %	0.00 %	0.00 %

Tabela 2: Porcentagem de gasto de tempo em cada função.

Os três primeiros testes apresentam uma quantidade muito limitada de dados, insuficiente para realizar uma análise percentual do tempo. Contudo, a partir do quarto teste, torna-se possível extrair várias conclusões. Primeiramente, tanto a operação de leitura quanto a de escrita do arquivo demonstram ser as menos dispendiosas, o que era esperado devido à sua complexidade linear. Da mesma forma, a identificação dos grupos consome muito pouco tempo em relação ao total, pois, como mencionado anteriormente, ocorre simultaneamente à escrita da saída, sem alocar recursos adicionais de memória, apenas acessando os valores de cada grupo. No cálculo das distâncias, percebe-se um peso significativo na complexidade do processo, como era previsto, uma vez que envolve dois loops de tamanho N. É válido ressaltar que o aumento na dimensão pode resultar em um leve acréscimo no tempo de execução, mas apenas se for um valor muito alto. Quanto à obtenção da MST, seu impacto no tempo total foi bastante reduzido; conforme nossa análise, isso é coerente, pois sua complexidade depende do número de pontos menos o número de grupos. Nota-se um aumento perceptível no tempo de execução no último teste, que apresenta um significativo aumento na quantidade de pontos. Por fim, a ordenação das distâncias consumiu mais da metade do tempo total. Essa função, proveniente de uma biblioteca do C, possui uma complexidade conhecida de $O(N \log N)$. No entanto, em seu pior cenário, pode tender a $O(N^2)$, o que justifica seu alto consumo de tempo em comparação às demais operações.

6- Conclusões finais

Com base na análise realizada, conclui-se que a implementação do algoritmo de espaçamento máximo utilizando o algoritmo de Kruskal juntamente com a estrutura union-find demonstrou-se eficiente e bem estruturada. A metodologia adotada permitiu uma abordagem sistemática na resolução do problema, destacando-se pela aplicação adequada de conceitos algorítmicos e estruturas de dados. Os testes empíricos realizados evidenciaram o desempenho satisfatório do algoritmo, com ênfase na otimização do tempo de execução. Dessa forma, este trabalho contribui para uma melhor compreensão do problema de espaçamento máximo e sua solução algorítmica, além de oferecer uma implementação robusta e eficaz para futuras aplicações.