**Trabalho Prática 2**

**Fecho Convexo**

**Eduardo Silveira Cezar Fernandes – 2021019424**

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais(UFMG)

Belo Horizonte – MG – Brazil

[eduardoscf1210@gmail.com](mailto:eduardoscf1210@gmail.com)

**1. Introdução**

Esta documentação lida com o problema de determinar qual é o menor fecho convexo possível dado um conjunto de pontos cartesianos. Para concretizar a funcionalidade proposta, o programa pode ser utilizado com 2 métodos principais, são eles: Scan de Graham e Marchar de Jarvis. Sendo que o primeiro deles pode ser executado com 3 métodos de ordenação diferente, o MergeSort, o InsertionSort e o BucketSort.

**2. Método**

**2.1. Configurações da máquina**

Sistema operacional:

Linguagem de programação: C++

Compilador: G++ / GNU Compiler Collection

Processador: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz 2.40 GHz

Memória RAM: 12,0 GB

**2.2. Estruturas de dados**

A estrutura de dados utilizada nesse trabalho foi o array. O uso dessa estrutura específica teve em vista armazenar e lidar com os pontos que o programa irá receber.

Há um array de pontos cartesianos alocado dinamicamente que irá receber todos os pontos do arquivo lido, pois dessa forma o programa será capaz de lidar com qualquer quantidade de pontos que o arquivo fornecer. Com este array, todas as operações do programa serão executadas a partir dele.

Já no Scan de Graham e no Marchar de Jarvis, foram criados array temporários dos pontos que pertencerão ao fecho convexo para então criar as retas formadas por esses pontos.

**2.3. Classes**

Na implementação desse trabalho, foram utilizadas 3 classes, PontoCartesiano, reta e FechoConvexo.

Como o programa recebe pontos cartesianos, é necessário ter um tipo para armazená-los, já na hora que o fecho convexo está sendo criado, primeiro são criadas as retas que vão compor o fecho convexo e em seguida essas retas são adicionadas.

A classe PontoCartesiano possui a maioria das funções e os métodos funcionam ao redor dela, ou seja, a ordenação é feita em um array de pontos.

A classe Reta possui os dois pontos que a formam e irá compor o fecho convexo.

A classe FechoConvexo irá ser formada a partir de retas e é objetivo principal do programa, ter esse fecho formado corretamente.

**2.4. Funções**

O código possui 6 funções principais, a mergeSort(), a insertionSort(), a bucketSort, o scanDeGraham(), o marcharDeJarvis() e por fim a avaliaTempo().

As funções mergeSort, insertionSort e bucketSort possuem o mesmo objetivo: ordenar o array de pontos lido em relação ao que o Scan de Graham pede. Ou seja, define o primeiro como aquele mais embaixo (menor coordenada Y) e em caso de mais de um ponto com a mesma coordenada Y, o desempate é feito com o menor X. Após isso, todos os demais pontos são ordenados de forma crescente em relação ao ângulo formado com esse ponto inicial. OBS: O ponto inicial é encontrado por uma função separada e este ponto é passado como parâmetro para esses métodos de ordenação.

A função Scan de Graham recebe o array já ordenado, com isso o algoritmo procede considerando cada um dos pontos do array ordenado em sequência. Para cada ponto, é determinado, se ao mover-se dos dois pontos anteriores para este ponto se forma uma "curva no sentido anti-horário" ou uma "curva no sentido horário". Se é uma "curva para esquerda", isto significa que o ponto de partida não faz parte do envoltório convexo e deve ser removido da pesquisa. Este processo continua ao longo do conjunto até que o conjunto dos três últimos pontos seja uma curva para direita. Assim que uma "curva no sentido anti-horário" é encontrada, o algoritmo salta para o próximo ponto do array ordenado.

Já o Marchar de Jarvis não precisa de ordenação. O algoritmo começa pelo ponto mais a esquerda, ou seja, aquele que possui menor coordenada X, depois continuamos percorrendo os pontos em sentido anti-horário. O ponto é atribuído ao fecho convexo se ele possui a maior orientação no sentido anti-horário em relação aos demais.

Em relação a função avaliaTempo, ela copia o array de pontos original 4 vezes, para não alterar o array original e usar cada array para testar o tempo de execução de cada função anterior.

Outras funções como orientacao(), calcularAngulo() e encontrarPrimeiroPonto são funções auxiliares para serem utilizadas por esses métodos principais.

**3. Análise de complexidade**

**mergeSort() – Complexidade de tempo:** A complexidade de tempo da função mergeSort é O(n log n), onde n é o número de elementos a serem ordenados. Essa complexidade é obtida porque o algoritmo divide recursivamente o array ao meio até que cada subarray tenha apenas um elemento, e então mescla esses subarrays em ordem crescente. A etapa de divisão tem uma complexidade logarítmica (O(log n)), pois o array é dividido ao meio a cada chamada recursiva. A etapa de mesclagem tem uma complexidade linear (O(n)), pois todos os elementos precisam ser comparados e mesclados. Portanto, a complexidade total do mergeSort é O(n log n).

**insertionSort() – Complexidade de tempo:** A complexidade de tempo da função insertionSort é O(n^2), onde n é o número de elementos a serem ordenados. Essa complexidade é obtida porque o algoritmo percorre o array a partir do segundo elemento até o último, e para cada elemento, realiza uma comparação e possivelmente uma troca com os elementos anteriores para colocar o elemento na posição correta.

**bucketSort() – Complexidade de tempo:** Como eu usei o bubbleSort para ordenar cada bucket, e ela possui uma complexidade O(n^2), a complexidade total do bucketSort será O(n) (colocando cada ponto no bucket correspondente) mais O((n/numBuckets)^2) + O(n). E como o número de buckets é próximo do número de elementos n, a complexidade de tempo da bucketSort será O(n) + O(n^2) + O(n) = O(n^2).

**scanDeGraham() – Complexidade de espaço:** A complexidade de espaço da função Scan de Graham depende da quantidade de pontos que o arquivo lido possui.Logo, a complexidade é O(tamanho), no caso, tamanho é o tamanho do array que aloca os pontos lidos.

**marcharDeJarvis() – Complexidade de espaço:** Assim como o scan de graham, a função do Marchar de Jarvis depende da quantidade pontos lidos, logo, sua complexidade de espaço é O(tamanho) também.

**avaliaTempo() – Complexidade de espaço:** A complexidade de espaço dessa função é proporcional ao tamanho (número de pontos) multiplicado pelo tamanho de “sizeof(PontoCartesiano)”, devido aos arrays pontos1, pontos2, pontos3 e pontos4.

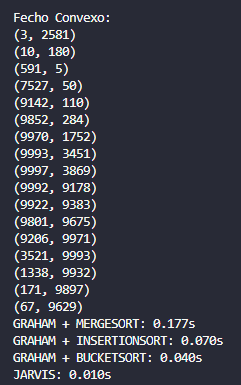
**4. Análise de robustez**

Para padronizar o código, todas as variáveis foram nomeadas de acordo com a sua função, além da indentar corretamente o código para ajudar na leitura. Também foram adicionados comentários esclarecedores por todo o programa.

Para tratar erros, o programa emite um aviso caso não seja possível abrir o arquivo de leitura corretamente. Já na hora de debugar o código, foi utilizado o GDB para entender passo a passo o que estava acontecendo com o programa. Também houve o uso do Valgrind para garantir que não estava acontecendo vazamento de memória.

**5. Análise experimental**

Utilizando o arquivo ENTRADA1000.txt fornecido pelo professor da matéria:



**6. Conclusão**

Dessa forma, o programa criado fornece corretamente o fecho convexo com a menor quantidade de pontos fornecidos pelo arquivo de entrada. Fornecendo também o tempo de execução de cada método assim como foi ensinado nas práticas da matéria.

Durante sua modelagem e execução, as maiores dificuldades encontradas foram a criação do método mergeSort, pois lida com recursividade e particularmente tenho dificuldade para compreender a execução de métodos recursivos e a criação do método bucketSort pois estava confuso em relação ao o que cada balde armazenaria.

**7. Bibliografia**

Márcio Costa Santos e Wagner Meira Jr. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte. <https://www.geeksforgeeks.org> <https://www.youtube.com/watch?v=SBdWdT_5isI>

<https://www.youtube.com/watch?v=nBvCZi34F_o&t=610s>

**8. Instruções de compilação e execução**

Diretamente do diretório TP2 utilize o comando “make run FILE=nomearquivo.txt”. No qual o nomearquivo.txt deve ser substituído pelo nome do arquivo que será lido e contém os pontos cartesiano.

Um exemplo de linha de comando para o uso do programa seria:

make run FILE=ENTRADA1000.txt