**Trabalho Pratico 2**

Fecho Convexo

**Kayque Meira Siqueira**

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

kayquesiqueira@ufmg.br

# Introdução

A ideia base do projeto proposto foi a implementação de um programa que seja capaz de encontrar o fecho convexo de um determinado conjunto de pontos no plano bidimensional. Um fecho convexo é, basicamente, o menor polígono convexo capaz de conter todos os outros pontos em seu interior, sendo formado pela ligação de alguns dos pontos do conjunto completo.

Para resolver o problema, foi criado uma estrutura capaz de receber um array (lista) de pontos, e encontrar o fecho convexo a partir do algoritmo *marchar de Jarvis* ou do algoritmo *scan de Graham.* Além disso, para resolver o problema, o programa ainda é capaz de ordenar o conjunto de pontos utilizando 3 algorítmos de ordenação: o merge-sort, o insertion-sort e o bucket-sort. Por fim, o programa imprime no terminal os valores dos pontos que formam o fecho convexo e o tempo que cada um dos algorítmos utilizando cada um dos métodos de ordenação levou para encontrar a solução.

Após encontrar o resultado, imprimi-lo no console e o programa terminar, se o usuário quiser visualizar graficamente o fecho convexo encontrado, há um tutorial de como gerar a imagem no final desta documentação.

# Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

# Estrutura de Dados

A implementação do programa teve como base a construção de uma estrutura de classe em c++ contém duas funções principais, a jarvisMarch e a grahamScan, que, internamente, percorrem o vetor de pontos passado por parâmetro e, a partir de funções de ordenação auxiliares, ordenam esse vetor e encontram o fecho convexo de acordo com os passos de cada algoritmo. Ao todo, a estrutura da classe possui 9 funções majoritárias, são elas:

- Duas funções principais que internamente utilizam as outras: jarvisMarch e grahamScan.

- Três funções de ordenação: mergeSort, insertionSort e bucketSort

- Quatro funções auxiliares:

1º - orientation: recebe 3 pontos por parâmetro e retorna a informação da inclinação das retas formada pela ligação deles, se são retas que está inclinada no sentido horário ou anti-horário.

2º - minXYPoint: recebe o array de pontos por parâmetro e retorna o ponto que possui a menor coordenada y e em caso de empate, aquela que tem a menor coordenada x.

3º - minXPoint: recebe o array de pontos por parâmetro e retorna o ponto que possui a menor coordenada x.

4º - merge: função auxiliar utilizada para o funcionamento do mergesort

A primeira coisa que o programa faz é tentar abrir o arquivo de texto que contém os pontos cujos quais o fecho deve ser encontrado. Após isso, armazena os pontos presentes no arquivo em um vetor de pontos interno. Por fim, é criada uma variável que instancia a estrutura principal e as funções jarvisMarch e grahamScan são chamadas recebendo como parâmetro o vetor de pontos, o tamanho desse vetor e, no caso da grahamScan, o tipo de ordenação que deve ser feito.

Além disso, vale ressaltar que o código também faz o uso de uma estrutura auxiliar chamada Pilha. As funções de resolução do algorítmo (a função jarvisMarch e a grahamScan) retornam uma pilha que contém os pontos encontrados que formam o fecho convexo. A função mais custosa de toda a estrutura é sem dúvidas a grahamScan com as escolhas de ordenação mergeSort ou bucketSort.

# Classes

A classe principal do programa é chamada de “ConvexHull”, que é responsável por conter as funções principais para resolução do problema abordado e possui 14 funções ao todo.

A classe “secundária” do programa é a classe da estrutura Pilha (extremamente importante), contendo todas as suas funções que serão posteriormente utilizadas pela classe principal para armazenar os pontos. A classe pilha possui 3 atributos, o tamanho da pilha, o vetor que armazenará os elementos da pilha e uma variável auxiliar que guardará o “index” referente ao elemento do topo da pilha, além de 7 funções para seu funcionamento, como a função “pop”, que remove o elemento do topo, e a função “push” que empilha um elemento.

Além disso, o programa conta com uma estrutura chamada “Dot” (ponto) que será responsável por guardar as coordenadas x e y de cada ponto lido do arquivo de entrada.

# 2.3 Leitura e Armazenamento

Para a leitura, o programa, com o auxílio da biblioteca fstream que realiza a abertura de arquivos para leitura. Ao iniciar a execução, o código, por meio dessa biblioteca, abre o arquivo passado como argumento na linha de comando onde estão os pontos.

Para ler as linhas desse arquivo é feito o uso da função nativa getline da própria linguagem de programação que obtém o conteúdo das linhas do arquivo no formato string.

Para cada linha do arquivo, o código cria uma variável que instancia uma estrutura do tipo “Dot” e armazena as informações das coordenadas x e y da linha, e depois armazena cada um desses pontos em um array de pontos. Essa variável array tem um tamanho inicial padrão de 100000.

# Analise Complexidade

Observando o programa como um todo, percebe-se que a funções auxiliares apresentam custo baixíssimo por nem possuírem loops, apenas poucas comparações de valor e operações matemáticas, como é o caso das funções “order”, “orientation”, “comparePoints”, “nextToTop”, “angulation” e “squareDist”. Já as funções “minXPoint”, e “minXYPoint” apresentam um custo de ordem O(n), em que n é o tamanho do array de pontos passado por parâmetro. Isso por conta de haver apenas 1 loop que itera sobre o vetor em cada uma das funções, além de poucas comparações.

Já as funções de ordenação apresentam as seguintes complexidades:

mergeSort: O(n \* log(n));

insertionSort: O(n) no melhor caso e O(n²) no pior caso e nos casos médios;

bucketSort: O(n²) no pior caso (quando todos os elementos estão em um único “bucket” só, o que não acontece no código) e O(n + k) no melhor caso e casos médios, onde “k” é o numero de buckets, podendo ser simplificada apenas para O(n + k);

Em que “n” é o número de pontos a serem ordenados.

Já a função jarvisMarch apresenta uma complexidade definida da seguinte forma:

- O(n): complexidade para encontrar o ponto mais a esquerda (de menor coordenada “x”).

- O(n \* h): “n” por conta da estrutura de repetição que irá iterar sobre o array de pontos, e “h” pelo loop interno que irá percorrer o array pontos que até então formam o fecho convexo.

Portanto, a complexidade total da função jarvisMarch é O(nh), pois T(n) = nh + n => O(nh). Cabe salientar ainda que quanto mais o número de pontos presentes no fecho convexo é próximo do número total de pontos, mais custoso fica esse método, podendo chegar até a ter uma complexidade quadrática O(n²).

Por fim, na função grahamScan, considerando os casos médios de cada algoritmo de ordenação e as partes mais custosas da resolução, temos a seguinte distribuição de complexidade:

-GrahamScan usando MergeSort: O(n \* log(n))

-GrahamScan usando InsertionSort: O(n² + n) => O(n²)

Isso por conta do algoritmo de ordenação insertionSort ser menos eficiente que o mergeSort para grandes conjuntos de dados.

-GrahamScan usando BucketSort: O(2n + k), simplificando, O(n + k)

Onde “k” é o número de buckets.

# Estratégias de Robustez

Foi feito o uso de asserts com o auxilio da biblioteca cassert que ao longo do programa foi checando questões relacionadas a:

-Consistência dos valores dos pontos nas pilhas

-Passagem de argumentos corretos para algumas funções

Além disso, o programa também previne erros de divisão por 0 e erros com o arquivo de entrada, alertando o usuário.

# Análise Experimental

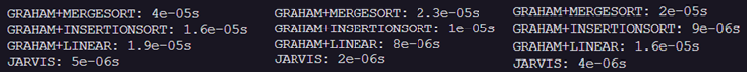
Para a analise experimental foi feito o uso do da biblioteca “time.h” para calcular o tempo de execução de cada um dos algoritmos.

Configurações da minha máquina:

Ryzen 7 5400U, 8gb RAM, 256gb SSD, Windows 10 Home.

Para realizar a análise, foi feito o uso de 3 arquivos padrões de entrada. Um deles com 10 pontos, outro com 100 pontos e o último com 1000 pontos. Para cada um dos arquivos, o código foi executado 3 vezes e foi feita uma média dos tempos obtidos desses 3 testes para cada uma das formas de resolução.

Primeiramente para o arquivo de 10 pontos obteve-se o seguinte resultado:



Fazendo a média dos valores, obteve-se:

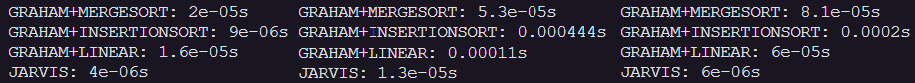
GRAHAM+MERGESORT: 0,000028 segundos.

GRAHAM+INSERTIONSORT: 0,000012 segundos.

GRAHAM+LINEAR: 0,000015 segundos.

JARVIS: 0,000004 segundos.

Agora para o arquivo de 100 pontos obteve-se o seguinte resultado:



Fazendo a média dos valores, obteve-se:

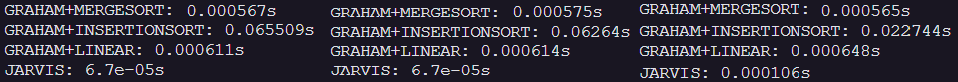
GRAHAM+MERGESORT: 0,000051 segundos.

GRAHAM+INSERTIONSORT: 0,00022 segundos.

GRAHAM+LINEAR: 0,000062 segundos.

JARVIS: 0,000008 segundos.

Por fim, para o arquivo de 1000 pontos obteve-se o seguinte resultado:



Fazendo a média dos valores, obteve-se:

GRAHAM+MERGESORT: 0,00057 segundos.

GRAHAM+INSERTIONSORT: 0,05 segundos.

GRAHAM+LINEAR: 0,00062 segundos

JARVIS: 0,00024 segundos.

Como pode-se perceber, em todos os casos, o algoritmo de resolução Jarvis foi o algoritmo mais rápido à solucionar o problema. Além disso, com 10 pontos, vê-se que o algoritmo graham scan utilizando o insertionSort foi o segundo mais rápido, entretanto, conforme os pontos aumentam, ele vai se tornando o mais lento das 3 formas de resolução do grahamScan. A medida que a quantidade de pontos crescem, o graham scan, usando o mergeSort, vai se tornando a segunda forma mais rápida de resolução, perdendo apenas para o Jarvis no quesito de tempo.

# CONCLUSÕES

Podemos concluir que o trabalho está consistente no que se diz respeito a sua complexidade em suas funções e o gráfico de tempo de execução das 4 formas ao longo do tempo. Está funcional em respeito da resolução do problema do fecho convexo e não está permitindo falhas com o uso de asserts e condicionais.

Houveram algumas dificuldades quanto ao estabelecimento de padrões para rodar o programa por conta de mudanças nas especificações do trabalho ao longo do tempo, mas, acredito que seguindo as instruções do tópico 8, os usuários não terão problemas em executar o código.

O programa está muito eficiente em questão de desempenho e corresponde com o que foi esperado inicialmente.

Ao longo do desenvolvimento percebe-se a importância das boas praticas de escrita de código como um dos principais alicerces que facilitaram a identificação de problemas e agilizaram a criação das correções necessárias. Além disso, cabe ressaltar a importância da escolha correta entre as diversas soluções de um problema, pois na maioria dos casos existem várias soluções, mas nem sempre a selecionada é a melhor para um certo propósito particular do usuário.

# Bibliografia

- Convex Hulls: Graham scan – Inside Code [(101) Convex hulls: Graham scan - Inside code - YouTube](https://www.youtube.com/watch?v=SBdWdT_5isI)

- Convex hulls: Jarvis march algorithm (gift-wrapping) - Inside code [(101) Convex hulls: Jarvis march algorithm (gift-wrapping) - Inside code - YouTube](https://www.youtube.com/watch?v=nBvCZi34F_o)

- Convex Hull Algorithm - Graham Scan and Jarvis March tutorial [(101) Convex Hull Algorithm - Graham Scan and Jarvis March tutorial - YouTube](https://www.youtube.com/watch?v=B2AJoQSZf4M&t=318s)

- Convex Hull using Jarvis’ Algorithm or Wrapping – GeeksForGeeks - [Convex Hull using Jarvis' Algorithm or Wrapping - GeeksforGeeks](https://www.geeksforgeeks.org/convex-hull-using-jarvis-algorithm-or-wrapping/)

- Convex Hull using Graham Scan – GeeksForGeeks - [Convex Hull using Graham Scan - GeeksforGeeks](https://www.geeksforgeeks.org/convex-hull-using-graham-scan/)

# Instruções para compilação e execução

Ao abrir o diretório que contém o código em seu editor de texto preferido ou no terminal, será necessário, primeiramente, colocar um arquivo de nome arbitrário no formato .txt que irá conter os pontos na raiz do projeto (na mesma pasta que contém o makefile). Após adicionar o arquivo de texto com os pontos, é necessário que seu sistema operacional tenha algum compilador da linguagem C++ instalado e que também tenha o “make” instalado.

Após isso, basta abrir a pasta raiz no terminal e executar o seguinte comando: “*make run fecho \*nome\_do\_arquivo\_txt\*.txt*”

Por exemplo, se o seu arquivo com os pontos se chamasse “points”, o comando seria: *make run fecho points.txt*

Ao executar esse comando, o makefile irá compilar os arquivos necessários e irá executar o programa considerando os pontos no arquivo txt de entrada. Os resultados aparecerão no terminal.

Para melhor visualização, há também a possibilidade de gerar uma imagem png com o desenho do fecho encontrado. Ao ser executado, o programa irá criar dois arquivos de texto: “scatter\_data.txt” e “lide\_data.txt”, que são arquivos necessários para gerar essa imagem caso o usuário queira gerá-la.

Para gerar a imagem é necessário seguir o seguinte procedimento:

1. Ter o python 3 instalado no sistema operacional e consequentemente o “pip”;
2. Ter instalado a biblioteca matplotlib pelo comando “pip install matplotlib”;
3. Executar o seguinte comando no terminal em que a pasta raiz do projeto está aberta: “python3 ordering.py”

Se tudo correr bem, um arquivo de imagem .png com o nome “result.png” será gerado e o usuário poderá ver o fecho graficamente.

**Cabe salientar que esta ultima parte de visualização gráfica é apenas um EXTRA do trabalho, não estava nos requisitos, mas é algo interessante que criei para incrementar a experiência, se alguma bonificação for dada, agradeço desde já!**