Trabalho Prático 2
Calculador de Fecho Convexo

Lorenzo Ventura Vagliano

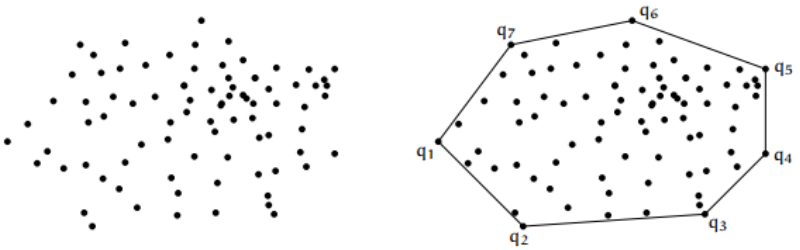
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

lorenzovagliano@dcc.ufmg.br - Matrícula: 2022035938

1 - Introdução

O problema proposto foi a implementação de algoritmos capazes de calcular o Fecho Convexo de um conjunto de pontos. O fecho convexo é um conceito importante na área da geometria computacional. Ele designa os vértices de um dado conjunto de pontos, que quando conectados formam o menor perímetro que engloba todos os pontos do conjunto de maneira a formar um polígono convexo, isso é, no qual todos os ângulos internos são menores que 180 graus.

Ilustração:



(a) Input. (b) Output.

Esse conceito possui diversas aplicações práticas, como o problema contextual trazido pela especificação do projeto, uma solução para a indústria têxtil que permita a maior economia de material possível.

Mais especificamente, foram testados 2 algoritmos conhecidos para a resolução desse problema, o Scan de Graham com 3 métodos de ordenamento diferentes e a Marcha de Jarvis.

Dessa maneira, foram avaliados 4 algoritmos:

- Marcha de Jarvis.
- Scan de Graham utilizando:
    - Merge Sort.
    - Insertion Sort.
    - Bucket Sort.

Ademais, foi desenvolvida uma **visualização gráfica** do fecho convexo, que nos permite ver a geração dos pontos, dos vértices do polígono e das retas que os conectam em tempo real.

Este recurso externo também fornece uma visualização didática do Scan de Graham e da Marcha de Jarvis.

2 - Método

O programa foi desenvolvido em linguagem C++ e compilado pelo compilador g++ versão 11.3.0. O método de Build foi feito utilizando GNU Make.

2.1 - Estruturas de Dados/Classes

Para a implementação do projeto, o tipo de dados essencial foi o Ponto 2D, que contém as coordenadas X e Y dos pontos dos quais é derivado o fecho convexo. Sobre esses pontos foram aplicados algoritmos para determinar seu produto vetorial, orientação e distância euclidiana.

Além disso, o segmento de reta, que consiste, basicamente, na relação entre dois pontos, foi utilizado ao ser avaliada a colinearidade dos pontos e para usos gráficos. O fecho convexo em si é representado apenas por uma coleção de pontos e pelo seu tamanho.

2.2 - Funções

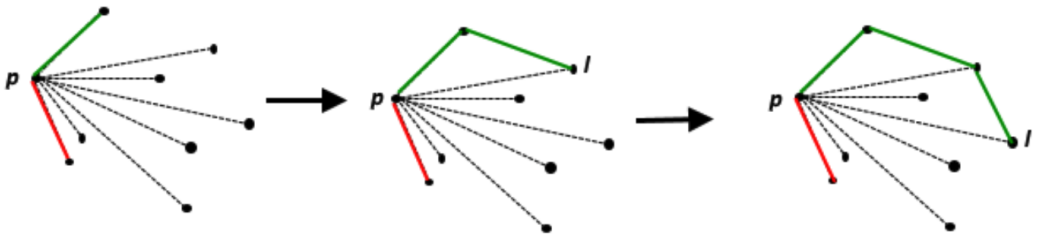
long int produtoVetorial3(const Ponto& A, const Ponto& B, const Ponto& C)

Calcula o produto vetorial de três pontos no plano 2D. O produto vetorial é uma operação que resulta em um vetor perpendicular ao plano formado pelos pontos A, B e C. Nessa implementação, o produto vetorial é calculado usando a fórmula determinante, onde as coordenadas dos pontos são utilizadas para obter o resultado. O primeiro termo da fórmula é dado pela diferença entre as coordenadas x de B e A multiplicada pela diferença entre as coordenadas y de C e A. O segundo termo é dado pela diferença entre as coordenadas y de B e A multiplicada pela diferença entre as coordenadas x de C e A. A diferença entre esses dois resultados fornece o valor do produto vetorial, que pode ser positivo, negativo ou zero, dependendo da orientação dos pontos em relação ao plano.

bool isEsquerda(const Ponto& A, const Ponto& B, const Ponto& P)

Verifica se o ponto P está à esquerda da linha definida pelos pontos A e B no plano 2D. Isso é determinado pelo sinal do produto vetorial calculado usando a função produtoVetorial3. Se o produto vetorial for maior que zero, significa que o ponto P está à esquerda da linha. Caso contrário, se for menor ou igual a zero, indica que o ponto P está à direita da linha ou colinear com ela. A função retorna um valor booleano, sendo true se P estiver à esquerda e false caso contrário.

Ponto* JarvisFecho(Ponto pontos[], long int n, long int& tamanhoFecho)



O algoritmo começa encontrando o ponto mais à esquerda entre os pontos fornecidos, como referência inicial. Em seguida, cria um vetor pontosFecho para armazenar os pontos do fecho convexo e inicializa o tamanho do fecho como zero.

Em um loop principal, o algoritmo avança de ponto em ponto ao redor do fecho convexo. Ele seleciona o próximo ponto q que está mais à esquerda em relação ao ponto atual p. Para isso, percorre todos os pontos e utiliza a função isEsquerda para verificar se o ponto i é à esquerda da linha definida pelos pontos p e q. Se um ponto i mais à esquerda é encontrado, ele é atualizado como o próximo ponto q.

Após encontrar o próximo ponto q, o algoritmo atualiza p com o valor de q e adiciona o ponto p ao vetor pontosFecho. O loop continua até que o próximo ponto q seja igual ao ponto mais à esquerda, indicando que o fecho convexo foi completado.

Ao final, a função retorna o ponteiro pontosFecho, que contém os pontos do fecho convexo. O tamanho do fecho tamanhoFecho é atualizado para refletir o número de pontos encontrados.

long int distanciaEuclidiana(const Ponto& A, const Ponto& B)

Calcula a distância euclidiana ao quadrado entre dois pontos no plano 2D. A função recebe dois pontos A e B como parâmetros e realiza a diferença das coordenadas x e y entre os pontos. Em seguida, calcula o quadrado das diferenças em x e y e retorna a soma desses quadrados, representando a distância euclidiana ao quadrado entre os dois pontos. O cálculo da distância ao quadrado é utilizado para evitar a necessidade de realizar a operação de raiz quadrada, o que pode ser computacionalmente mais eficiente em certos contextos.

bool compararPontos(const Ponto& A, const Ponto& B, const Ponto& reference)

Verifica se dois pontos A e B são colineares em relação a um ponto de referência. Ela calcula o produto vetorial entre os vetores formados pelos pontos A e B em relação ao ponto de referência. Se o produto vetorial for igual a zero, indica que os pontos são colineares e, nesse caso, a função utiliza a distância euclidiana entre cada ponto e o ponto de referência para determinar qual ponto está mais próximo. Caso contrário, o resultado do produto vetorial é utilizado para determinar se o ponto A está à esquerda ou à direita do vetor B - reference. A função retorna true se A estiver à esquerda e false se estiver à direita.

void merge(Ponto pontos[], long int esquerda, long int meio, long int direita, const Ponto& reference)

Implementa o algoritmo de mesclagem (merge) utilizado no algoritmo de ordenação merge sort. Ela recebe um array de pontos pontos, os índices de esquerda, meio e direita que delimitam as partes a serem mescladas, e um ponto de reference utilizado para comparação. A função cria dois arrays temporários, arrEsq e arrDir, e copia os elementos correspondentes da parte esquerda e direita do array original para esses arrays. Em seguida, realiza a mesclagem propriamente dita, comparando os pontos de arrEsq e arrDir utilizando a função compararPontos e colocando o ponto correspondente no array pontos. Após a mesclagem, os elementos restantes de arrEsq e arrDir são copiados para pontos. Por fim, os arrays temporários são liberados da memória. Esse processo de mesclagem é utilizado no merge sort para ordenar o array de pontos com base em uma referência fornecida.

void mergeSort(Ponto pontos[], long int esquerda, long int direita, const Ponto& reference)

A função é uma implementação do algoritmo de ordenação "merge sort" para ordenar um array de pontos. Ela utiliza a técnica de divisão e conquista, dividindo o array em partes menores, ordenando-as recursivamente e, em seguida, mesclando as partes ordenadas para obter o array final ordenado. A função recebe como parâmetros o array de pontos pontos, os índices esquerda e direita que definem o intervalo a ser ordenado, e o ponto de referência reference utilizado pela função compararPontos durante a mesclagem.

A função verifica se esquerda é menor que direita, o que garante que existam pelo menos dois elementos para ordenar. Em seguida, calcula o índice do elemento do meio do intervalo com a fórmula (esquerda + direita) / 2. A função é chamada recursivamente para ordenar a metade esquerda do intervalo, passando esquerda como início e meio como fim. Em seguida, é chamada novamente para ordenar a metade direita do intervalo, passando meio + 1 como início e direita como fim. Após as chamadas recursivas, a função merge é chamada para mesclar as duas partes ordenadas em um único array ordenado.

Esse processo de divisão e conquista continua até que o intervalo contenha apenas um elemento, quando a recursão atinge seu caso base. Nesse caso, a função retorna e a mesclagem das partes ordenadas começa. A combinação das chamadas recursivas e a mesclagem subsequente garantem que todos os elementos sejam corretamente ordenados.

void insertionSort(Ponto pontos[], long int n, const Ponto& reference)

Implementa o algoritmo de ordenação por inserção, que percorre um array de pontos pontos e insere cada elemento em sua posição correta dentro da porção já ordenada do array. O algoritmo começa iterando a partir do segundo elemento até o último elemento do array. Para cada elemento, ele é armazenado em uma variável temporária chamada chave, e em seguida, é comparado com os elementos anteriores da porção ordenada. O loop interno continua enquanto j é maior ou igual a zero e a função compararPontos retorna falso para a

comparação entre pontos[j] e chave, com base na referência fornecida. Enquanto essa condição for satisfeita, os elementos maiores são deslocados uma posição para a direita para abrir espaço para a inserção da chave. Após encontrar a posição correta, a chave é inserida no lugar apropriado. O processo é repetido para cada elemento restante no array até que todos estejam ordenados. Esse algoritmo é eficiente para ordenar pequenos conjuntos de elementos ou quando o array já está parcialmente ordenado.

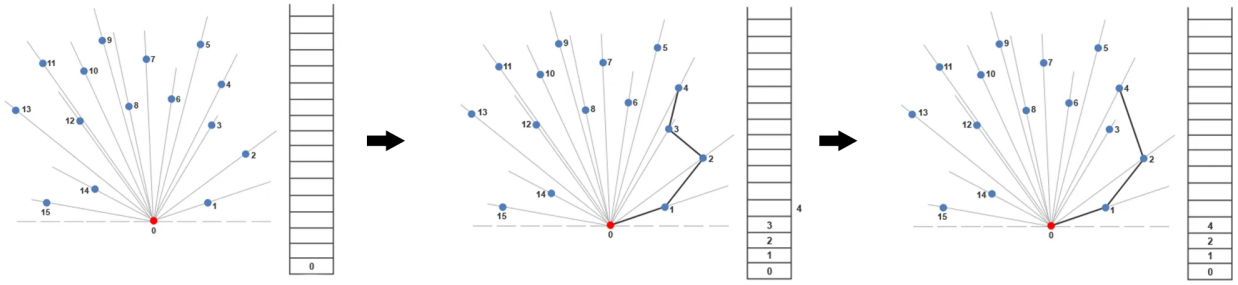
void bucketSort(Ponto* points, long int size)

Implementa o algoritmo de ordenação por buckets, que é um algoritmo de ordenação distributiva. O algoritmo divide o intervalo dos valores de coordenadas x em vários "buckets" com base em um tamanho de intervalo pré-definido. Inicialmente, ele determina o valor mínimo e máximo de coordenadas x nos pontos de entrada para calcular o número necessário de buckets. Em seguida, cria os buckets vazios e, em seguida, distribui os pontos nos buckets apropriados com base em seus valores de coordenadas x. Cada ponto é atribuído a um bucket de acordo com a fórmula (ponto.x - minX) / bucketRange, onde minX é o valor mínimo de coordenadas x e bucketRange é o tamanho do intervalo do bucket. Depois disso, os pontos dentro de cada bucket são classificados usando o algoritmo de ordenação por inserção. Em seguida, os pontos classificados em cada bucket são concatenados em uma única sequência para produzir a lista final de pontos ordenados.

Ponto* MergeConvexHullGraham(Ponto pontos[], long int n, long int& tamanhoFecho)

Ponto* InsertConvexHullGraham(Ponto pontos[], long int n, long int& tamanhoFecho)

Ponto* BucketConvexHullGraham(Ponto pontos[], long int n, long int& tamanhoFecho)



Os 3 algoritmos começam selecionando um ponto de referência a partir dos pontos fornecidos. Para isso, ele procura o ponto com a menor coordenada y (ou, em caso de empate, o ponto com a menor coordenada x). Essa escolha é importante, pois esse ponto será o ponto de partida para a construção do fecho convexo.

Após determinar o ponto de referência, o algoritmo procede ordenando os pontos restantes em relação ao ângulo que eles formam com o ponto de referência. Essa ordenação pode ser feita utilizando qualquer um dos 3 métodos de ordenamento implementados. Nela, cada ponto é inserido em sua posição correta na sequência já ordenada. Essa etapa é crucial para garantir

que os pontos estejam dispostos em sentido horário ou anti-horário ao redor do ponto de referência.

Com os pontos ordenados, o algoritmo prossegue para a etapa final de construção do fecho convexo. Ele percorre os pontos ordenados e verifica se eles formam uma curva à esquerda em relação aos dois pontos anteriores já incluídos no fecho convexo. Caso a curva seja à esquerda, o ponto é adicionado ao fecho convexo. Caso contrário, o ponto anterior é removido do fecho convexo até que seja possível adicionar o ponto atual. Esse processo é repetido até que todos os pontos tenham sido examinados.

No final da execução do algoritmo, o fecho convexo resultante é retornado como um array de pontos.

3 - Análise de Complexidade

Jarvis:

Complexidade de Tempo: O algoritmo possui uma complexidade de tempo de O(nh), onde n é o número de pontos de entrada e h é o número de pontos no fecho convexo. No pior caso, quando o fecho convexo é formado por todos os pontos, a complexidade é O(n^2), mas em média, a complexidade é **O(nh)**.

Complexidade de Espaço: **O(n)**, onde n é o número de pontos de entrada, pois cria um novo array para armazenar o fecho convexo

Graham + Merge Sort

Complexidade de Tempo: O algoritmo consiste em três etapas principais: encontrar o ponto mais baixo, ordenar os pontos em ordem polar em relação ao ponto mais baixo e construir o fecho convexo usando uma pilha. A etapa de ordenação utiliza o algoritmo de merge sort, que possui uma complexidade de tempo de O(n log n). As outras etapas do algoritmo têm complexidade de tempo linear em relação ao número de pontos. Portanto, a complexidade total é dominada pelo merge sort e é **O(n log n)**.

Complexidade de Espaço: **O(n)**, onde n é o número de pontos de entrada, pois cria um novo array para armazenar o fecho convexo

Graham + Insertion Sort:

Complexidade de Tempo: A complexidade de tempo do algoritmo de inserção é O(n^2) no pior caso, pois precisa percorrer a lista de pontos várias vezes para inserir cada ponto na posição correta. Portanto, a complexidade total é dominada pelo algoritmo de inserção e é **O(n^2)**.

Complexidade de Espaço: **O(n)**, onde n é o número de pontos de entrada, pois cria um novo array para armazenar o fecho convexo

Graham + Bucket Sort:

Complexidade de Tempo: A complexidade de tempo do Bucket Sort depende do algoritmo de ordenação utilizado para ordenar os elementos em cada bucket. Neste caso, o algoritmo de inserção é usado para ordenar os pontos em cada bucket. Portanto, a complexidade de tempo total no pior caso é **O(n^2)**. Entretanto, devido à divisão em baldes, a complexidade deve ser significativamente menor devido à baixa probabilidade do pior caso.

Complexidade de Espaço: **O(n)**, onde n é o número de pontos de entrada, pois cria um novo array para armazenar o fecho convexo e diversos arrays configuram os buckets.

4 - Estratégias de Robustez

A fim de prover mais robustez ao sistema. Além de seguir boas práticas de desenvolvimento, diversos casos foram tratados.

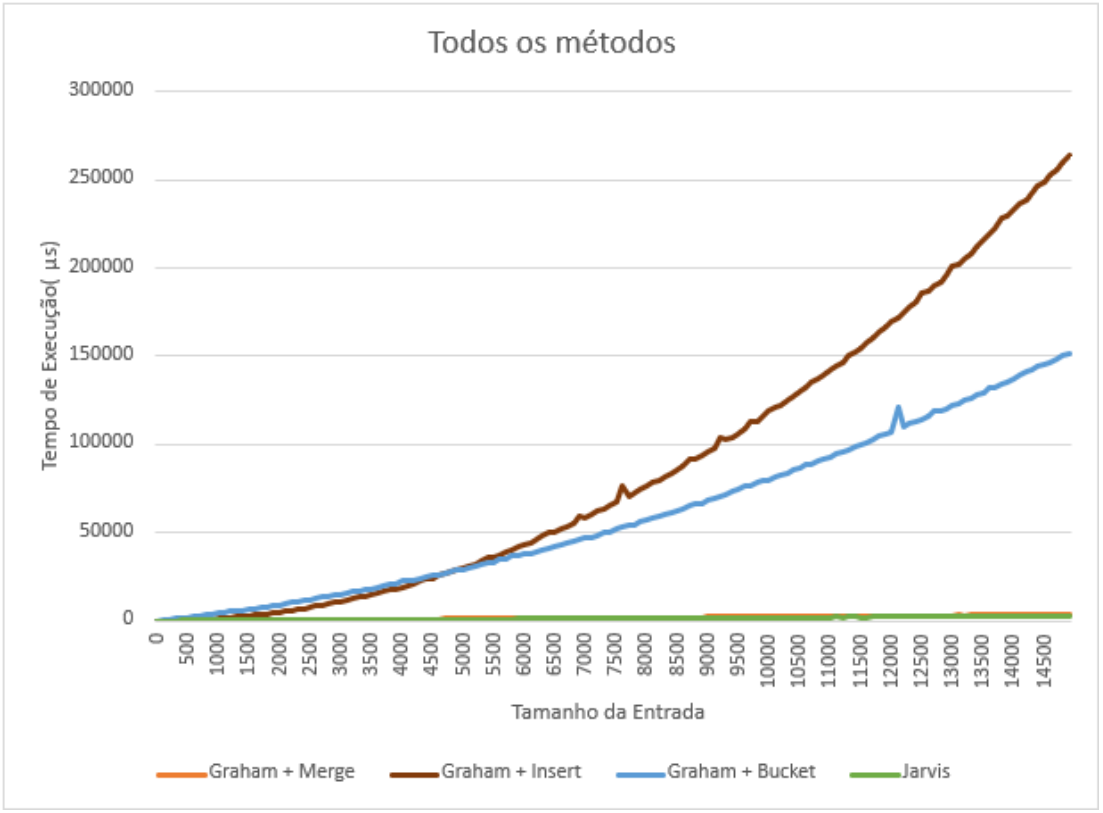
Primeiramente, todas as funções de geração de fecho convexo são capazes de detectar se o conjunto de pontos oferecidos como entrada, de fato possui um fecho convexo, que requer que existam pelo menos 3 pontos no plano. Caso contrário, o usuário é alertado ao executar a função.

Além disso, pelo fato do programa utilizar arquivos de texto para inserir os pontos a serem analisados, precauções foram tomadas para caso o arquivo não seja encontrado, caso o formato das entradas no arquivo esteja diferente do padrão fornecido pela especificação ou caso o número máximo de entradas seja ultrapassado.

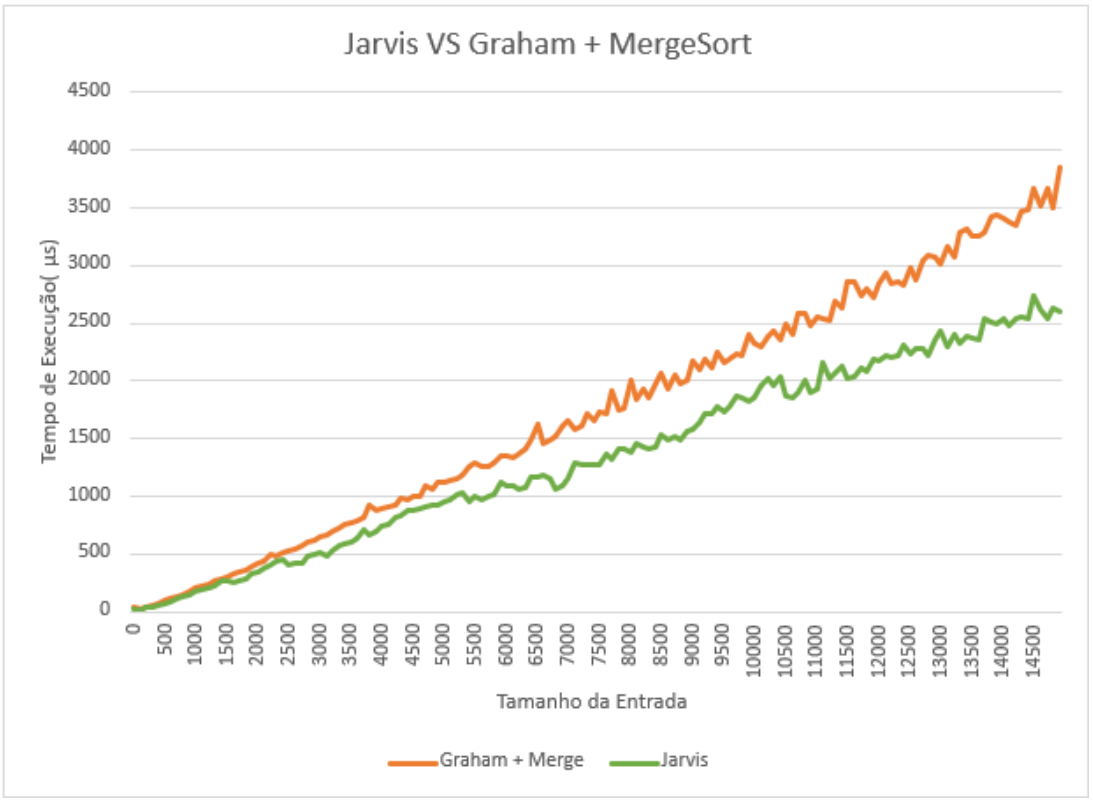
Ao se utilizar um comando de execução errado, o usuário também é alertado.

5 - Análise Experimental

Para avaliar a performance dos 4 algoritmos principais, foi criado um programa auxiliar, que gera conjuntos de pontos aleatórios (com coordenadas de 0 até 10,000) gradativamente maiores (contendo de 0 até 15,000 pontos). A partir disso, foram criados gráficos para avaliar a evolução do tempo de execução com a entrada.

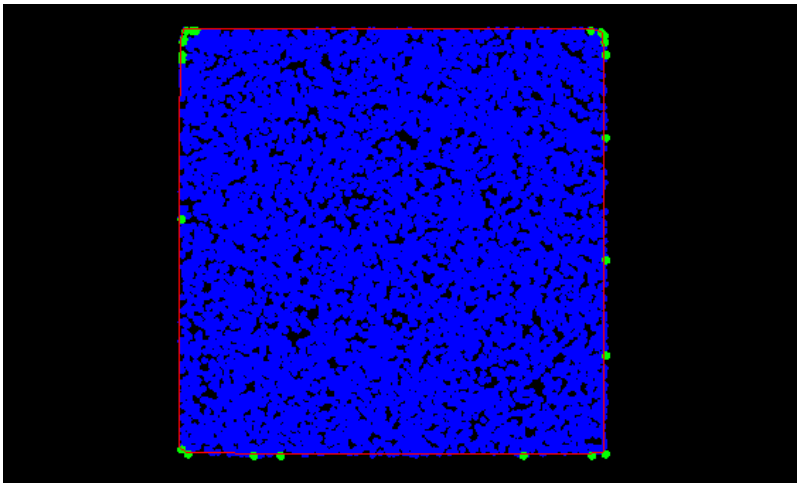


Como a implementação usando o Jarvis e o Graham + Merge Sort foram extremamente mais eficientes, foi gerado outro gráfico auxiliar para melhor visualização:



Analisando essas demonstrações, é possível avaliar que nossa análise teórica quanto ao Insertion Sort e o Bucket Sort se provaram certeiras, com o Bucket Sort sendo mais eficiente devido ao seu caso médio mais eficaz.

O método de Jarvis se provou consistentemente mais eficiente do que o de Graham + Merge Sort. Isso pode ser explicado pelo fato de que o pior caso do método de Jarvis ocorre quando o fecho convexo é formado por todos os pontos, o que raramente ocorre devido à aleatoriedade dos dados. Enquanto isso, o Merge Sort se prova cada vez menos eficaz com o crescimento da entrada, especialmente quando log n se torna maior do que 1. Podemos ilustrar essa hipótese quanto ao Jarvis utilizando a demonstração gráfica do fecho convexo em um caso contendo 10,000 pontos aleatórios (em azul), com os vértices do polígono convexo (em verde) muito menos numerosos.



6 - Conclusões

Por intermédio desse projeto, foi possível experimentar com a implementação do Scan de Graham e da Marcha de Jarvis, além de todos os métodos de ordenação utilizados pelo Graham. Não só isso, mas com a implementação de um modo de visualização, foi possível ilustrar o problema e averiguar hipóteses.

A partir desses esforços, pôde-se avaliar a complexidade de tempo e memória de cada algoritmo, levando à conclusão de que o Jarvis e o Graham + Merge Sort são disparadamente mais eficientes, elucidando o perigo da complexidade quadrática para grandes entradas.

7 - Bibliografia

Convex Hull Algorithms, The University of Tennessee Knoxville (Atwater & Dojcsak, 2020).
Convex Hull Algorithm Graham Scan and Jarvis March tutorial, Youtube (Stable Sort, 2020).
Fecho Convexo Planar, IME-USP (Pina, Shintate & Cordeiro).
Slides Estruturas de Dados, UFMG (Chaimowicz & Prates, 2019).
SFML Documentation: https://www.sfml-dev.org/learn.php (Open Source).

8 - Instruções para Compilação e Execução

Primeiramente, deve-se extrair os conteúdos do ZIP do projeto em uma pasta.

Antes de compilar o programa, é necessário ter instalado o GCC, o GNU Make e a **biblioteca gráfica SFML**, que é **obrigatória** para o funcionamento do programa.

Antes de instalar qualquer coisa, para atualizar as informações de pacote, utilize o comando:

```
sudo apt-get update
```

Para instalar a SFML, após abrir um terminal no diretório do projeto, utilize o comando:

```
make install
```

Se for preferido ao make install, pode-se utilizar o comando:

```
sudo apt-get install libsfml-dev
```

Para compilar e executar o programa, utilize o seguinte comando:

```
make run ENTRADA=<arquivo de entrada> VIDEO=<sim ou nao>
```

- <arquivo de entrada> deve ser substituído pelo nome completo do arquivo de texto contendo os Pontos de entrada, registrados como descrito na especificação do projeto. **Esse arquivo deve estar contido na raiz do projeto**.
- <sim ou nao> deve ser substituído por “sim” ou “nao”, sendo essa a opção, ou não, por gerar a visualização do algoritmo.

Como detalhado na especificação, **a plataforma de referência é o Linux Ubuntu**. Qualquer diferença de setup devido a um ambiente diferente é responsabilidade do avaliador.

Limitações:

- Número máximo de pontos: 1,000,000.
- Coordenada máxima de pontos: até 10,000 em uma das coordenadas(x ou y), até 100,000 na outra, ou vice-versa.