Trabalho Prático 2

Fecho Convexo

João Victor Evangelista Cruz

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

joaovecruz@ufmg.br

1. Introdução

O problema proposto foi implementar um algoritmo capaz de gerar o menor polígono convexo que encapsule todos os pontos definidos em um plano cartesiano, também chamado de Fecho Convexo, para auxiliar na produção de peças de tecido de uma indústria têxtil. Para isso, há duas possibilidades: o Scan de Graham e o Marchar de Jarvis. Por isso, é necessário implementar os dois métodos e testar qual dos dois tem a melhor performance. Como o Scan de Graham utiliza métodos de ordenação para ordenar os pontos antes de definir quais formam o fecho, foi definido para implementar o Scan de Graham utilizando três diferentes métodos de ordenação: Mergesort, Insertion Sort e Bucket Sort, além do Marchar de Jarvis.

Por isso, para a solução do problema, é necessário a utilização de estruturas de dados para a representação dos pontos e do fecho convexo, além de estruturas auxiliares e funções para a ordenação dos pontos e para a obtenção do fecho convexo a partir dos dois métodos apresentados.

Scan de Graham:

O método funciona da seguinte maneira: Primeiramente encontra-se o ponto mais para baixo (menor valor de y) do conjunto de pontos que se quer obter o fecho convexo, e a partir dele, ordena o resto dos pontos da seguinte maneira: O ponto p1 vem antes do ponto p2 na ordenação se p2 tiver ângulo polar maior (no sentido anti-horário) do que p1.

Após a ordenação, verifica se há pontos que possuem ângulos iguais, e caso tenha, retira-se todos menos o que tiver a maior distância em relação ao ponto mais abaixo. Após isso, verifica se há mais de três pontos restantes. Caso não tenha, é impossível fazer o fecho, já que um polígono deve ter pelo menos três lados, e com isso, três pontas também.

Caso tenha pelo menos três pontos, vai inserindo os pontos em uma pilha e retira-se os pontos que fazem uma curva para a direita no plano cartesiano. Os pontos restantes formam o fecho convexo.

Marchar de Jarvis:

Inicialmente, verifica se o conjunto de pontos possui pelo menos 3 pontos. Caso não tenha, não há como fazer o fecho. Caso tenha, encontra-se o ponto mais à esquerda do plano e define ele como o ponto p. Após isso vai andando pelos pontos procurando o ponto com o menor ângulo polar no sentido anti-horário em relação a p. Quando encontra-se esse ponto, armazena ele e procura atualiza p para esse ponto. E assim vai até voltar ao ponto inicial. Os pontos armazenados ao fim do processo formam o fecho convexo.

2. Método

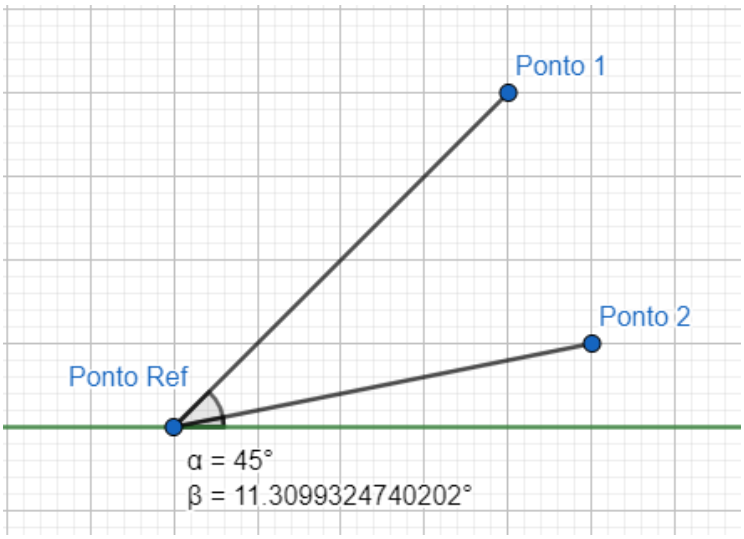
O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1 Estruturas de Dados

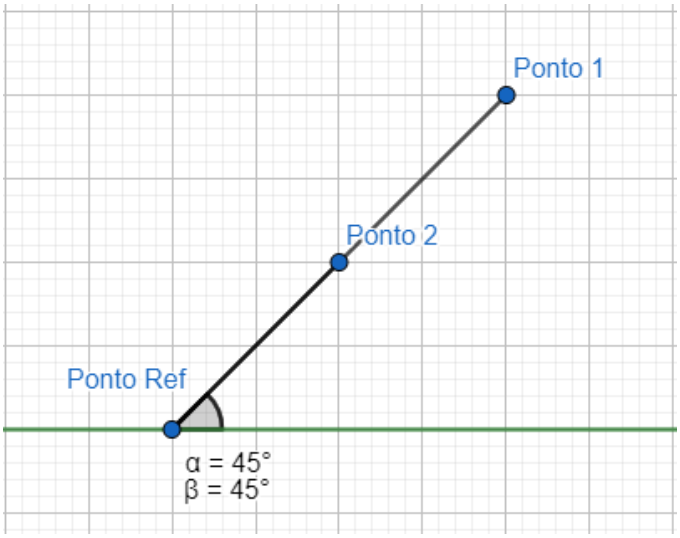
A implementação do programa teve como base as estruturas de dados Pilha e Lista, além da utilização de vetores.

A Pilha é utilizada no Scan de Graham para armazenar os elementos do TAD Ponto, que são gerados pelo método e que no final, formam o fecho convexo.

Já a Lista é utilizada para auxiliar o método de ordenação Bucket Sort, e também armazena elementos do TAD Ponto. A lista é criada utilizando um ponto que é passado como parâmetro e serve de referência para a inserção dos outros pontos na lista. Os pontos colocados na lista já são inseridos ordenadamente, de forma que é calculado um ângulo entre o próprio ponto, o ponto de referência da lista e a linha da coordenada x do ponto de referência, e os pontos que tiverem um ângulo menor são posicionados antes dos pontos com ângulos maiores. Se o ângulo calculado for igual, pontos que tiverem distâncias menores distâncias em relação ao ponto de referência são posicionados antes dos pontos que possuem distâncias maiores.



Como mostrado nesse exemplo acima, se os pontos 1 e 2 forem inseridos na lista, o ponto 2 é posicionado antes do 1, pois ele possui um ângulo em relação ao ponto de referência menor do que o ângulo do ponto 1. (α é o ângulo do ponto 1 e β é o ângulo do ponto 2).



Já nessa configuração, os pontos 1 e 2 possuem ângulos iguais, então, eles são posicionados na lista de acordo com suas distâncias em relação ao ponto de referência. Como o ponto 2 está mais perto do que o ponto 1, o ponto 2 ficaria antes do ponto 1.

Já os vetores foram utilizados para armazenar os pontos que são passados para o programa pelo arquivo de entrada, além de ser parâmetro para os métodos de ordenação e os métodos de Graham e Jarvis, sendo que no de Jarvis, é retornado um vetor contendo os pontos que formam o fecho convexo.

2.2 Tipos Abstratos de Dados

Os TADs criados para esse programa foram o Ponto e o FechoConvexo, além das classes criadas para a Lista e Pilha, que já foram citadas, e a classe Célula que auxilia o funcionamento dessas duas classes,

armazenando os elementos dessas estruturas de dados e servindo como apontadores para os elementos posteriores.

O TAD Ponto possui membros do tipo inteiro x e y, que representam as coordenadas do ponto. Além disso, possui funções de Get e Set para obter e definir essas coordenadas. Possui também funções auxiliares para a impressão das coordenadas do ponto na tela, além de funções que calculam coisas importantes para o funcionamento dos métodos de obtenção do fecho convexo, como o ângulo e a distância.

O TAD Ponto também é usado no programa para a representar vetores obtidos através de outros pontos, já que vetores também são representados por coordenadas. Esses vetores são importantes para a obtenção do ângulo, através da função que calcula o produto escalar dos vetores envolvidos no cálculo.

Já o TAD FechoConvexo possui como membros um array de pontos que armazena os pontos que formam o fecho convexo, além de um membro do tipo inteiro que mostra quantos pontos formam o fecho. Além disso, possui funções que aplicam os métodos de Graham e de Jarvis para obter o fecho convexo e uma função para imprimir na tela os pontos do fecho.

2.3 Funções

Ponto:

Ponto::Ponto(int n, int m): Construtor do TAD Ponto, que recebe como parâmetro dois inteiros que definem os valores de x e y

Ponto::Ponto(): Construtor padrão do TAD Ponto, que não recebe parâmetro e define o valor de x como 1000 e y como -1000.

int Ponto::Getx(): Função auxiliar que retorna um inteiro com o valor de x.

int Ponto::Gety(): Função auxiliar que retorna um inteiro com o valor de y.

void Ponto::Setx(int n): Função auxiliar que define o valor de x como n.

void Ponto::Sety(int n): Função auxiliar que define o valor de y como n.

void Ponto::Imprime(): Função que imprime na tela as coordenadas x e y do ponto.

int Ponto::ProdutoEscalar(Ponto p): Função que retorna o produto escalar do próprio vetor com o vetor p. (Vetores representados pelo TAD Ponto, como já explicado anteriormente).

double Ponto::Angulo(Ponto b): Função que tem um retorno do tipo double e representa o ângulo formado pelo próprio ponto, pelo ponto b e pela linha da coordenada x do ponto, como já foi explicado anteriormente.

FechoConvexo:

FechoConvexo::FechoConvexo(Ponto pontos[], int n): Contrutor do TAD FechoConvexo que recebe como parâmetro um vetor de pontos e um inteiro que representa o tamanho do vetor. No construtor, é chamada a função Graham que obtem o fecho convexo a partir do método de ordenação Mergesort e os pontos que formam o fecho são armazenados no objeto, além de definir o tamanho do fecho como o número de pontos que formam o fecho.

void FechoConvexo::Imprime(): Função que chama a função Imprime do TAD Ponto em todos os pontos do fecho e imprime na tela esses pontos.

Pilha FechoConvexo::Graham(Ponto pontos[], int n, int tipo): Função que recebe como parâmetro um vetor de pontos, um inteiro que indica o tamanho do vetor e um outro inteiro de nome “tipo” que indica qual método de ordenação será utilizado. Se tipo for 0, o Mergesort é utilizado, se for 1, utiliza-se o Insertion Sort e se for 2, utiliza o Bucket Sort. A função utiliza o método Scan de Graham para obter o fecho convexo e retorna uma Pilha com os pontos que formam o fecho.

Ponto * FechoConvexo::Jarvis(Ponto pontos[], int n): Função que recebe como parâmetro um vetor de pontos e um inteiro que indica a quantidade de pontos que há no vetor e utiliza o método Marchar de Jarvis para obter o fecho convexo. A função retorna um vetor contendo os pontos do fecho.

MetOrd:

Nos arquivos MetOrd.cpp e MetOrd.hpp há funções que auxiliam e fazem a ordenação dos elementos, necessário para o Scan de Graham.

int orientation(Ponto p, Ponto q, Ponto r): Função que recebe três pontos e retorna um valor inteiro de acordo com a orientação desses pontos: Retorna 0 se os pontos forem colineares, 1 se tiverem orientação no sentido horário e 2 se tiverem orientação no sentido anti-horário.

void swap(Ponto &p1, Ponto &p2): Função que recebe dois pontos e troca os dois pontos, fazendo com que p1 passe a ser p2 e vice-versa.

void Mergesort(Ponto *v, int n, Ponto p): Função que recebe como parâmetro um vetor de pontos a serem ordenados, um inteiro com o tamanho do vetor e um ponto que serve como referência para a ordenação. A função chama a função auxiliar sort e ordena o vetor de forma que o ponto com menor ângulo polar no sentido anti-horário em relação a p fique antes dos pontos de maior ângulo. Se o ângulo for igual, o que tiver a menor distância de p, fica antes dos que tem maior distância.

void sort(Ponto *v, Ponto *b, int i, int f, Ponto p): Função recursiva que, dado um vetor de pontos v e dois inteiros i e f, ordena o vetor v[i..f] de acordo com a orientação dos pontos. O vetor b é utilizado internamente durante a ordenação.

void merge(Ponto *v, Ponto *b, int i, int m, int f, Ponto p): Função que, dado um vetor v e três inteiros i, m e f, sendo v[i..m] e v[m+1..f] vetores ordenados, coloca os elementos destes vetores, ordenadamente, no vetor em v[i..f].

void Insercao(Ponto *v, int n, Ponto p): Função que recebe como parâmetro o vetor de pontos v, um inteiro com o tamanho do vetor e o ponto p que serve de referência para a ordenação. O vetor v é ordenado pelo método de ordenação de Inserção, que consiste em substituir os elementos do vetor até encontrar a posição correta do elemento, e fazer isso com todos os elementos do vetor. Utiliza a mesma regra de ordenação do Mergesort.

void Bucket(Ponto *v, int len, Ponto p): Função que recebe como parâmetro o vetor de pontos v, um inteiro com o tamanho do vetor e ponto p que serve de referência para a ordenação. Utiliza a mesma regra de ordenação dos métodos anteriores. Utiliza o método de ordenação Bucket Sort, que utiliza listas encadeadas e divide os elementos do vetor nessas listas de acordo com o ângulo que os pontos fazem com o ponto p. Nessas listas, eles são inseridos ordenadamente e depois voltam ordenados para o vetor.

Pilha:

A classe Pilha possui os métodos padrão de uma pilha encadeada, com a exceção da função NextToTop:

Ponto Pilha::NextToTop(): Função auxiliar que retorna o ponto abaixo do topo da lista.

Lista:

A classe Lista possui os métodos padrão de uma lista encadeada, com a exceção das funções Insere e PreencheVetor:

void Lista::Insere(Ponto pnt): Função que recebe o ponto pnt que vai ser inserido na lista. O ponto já é inserido ordenadamente, utilizando a mesma regra de ordenação dos métodos de ordenação, utilizando o ângulo.

void Lista::PreencheVetor(Ponto *v, int i, int len): Função que recebe um vetor de pontos v, um inteiro i e um inteiro len que indica a quantidade de elementos da lista. O vetor v é preenchido pelos elementos da lista a partir da posição v[i].

3. Análise de Complexidade

A complexidade do programa é medida pela complexidade dos algoritmos de Graham e de Jarvis, e a de Graham está ligada à complexidade dos algoritmos de ordenação.

Insertion Sort

A complexidade do Insertion Sort é n-1 = O(n) no melhor caso, quando os elementos já estão ordenados, e $(n^2 - n)/2 = O(n^2)$ no pior caso, quando os elementos estão na ordem reversa.

Mergesort

O mergesort executa o mesmo número de comparações, independente da entrada. Por isso, o pior caso é igual ao melhor caso. O custo da função merge é linear (n), logo:

Para n = 1, T(1) = 0;

Para n > 1, T(n) = T(n/2) + n = O(n log n);

Porém, como deve-se ter um vetor auxiliar para o funcionamento do mergesort, o algoritmo requer espaço extra proporcional a n, o que aumenta seu custo de memória.

Bucket Sort

Como o Bucket utiliza Listas, a complexidade de inserir n elementos nas listas, com cada um tendo um custo de inserção de n/k, é de $n*n/k = O(n^2/k)$. Como há seis listas no Bucket Sort implementado, o custo é de $O(n^2/6) = O(n^2)$.

Além disso, há o custo de memória proporcional a n pela memória utilizada nas listas.

Scan de Graham

O método tem complexidade de: n (para obter o ponto com menor y) + c (Complexidade do método de ordenação) + n (verificação dos pontos de mesmo ângulo) + m-3 (verificação se algum ponto faz curva para a direita), em que m <= n. Logo a complexidade é de n + c. Para o Graham utilizando o Insertion Sort, no melhor caso, temos O(n+n) = O(n). No pior caso, temos $O(n + n^2) = O(n^2)$. Nesse caso, não há utilização de memória adicional. Para o Mergesort, sempre tem complexidade de O(n+ n log n) = O(n log n). Há também a utilização de memória extra devido ao Mergesort. Utilizando o Bucket Sort, tem-se o custo $O(n+n^2) = O(n^2)$ além da memória adicional das listas, proporcional a n.

Marchar de Jarvis

O método tem complexidade de: n (para obter o ponto mais à esquerda) + m (iteração pelos pontos da borda até chegar no ponto inicial) * n (verifica qual é o ponto mais no sentido anti-horário), em que m<=n. Por isso a complexidade é O(mn), que pode ser próximo de O(n), quando há muitos pontos no conjunto inicial e o fecho não tem tantos pontos quanto n, no melhor caso, e o pior caso é quando m = n, e todos os pontos formam o fecho convexo, tendo complexidade $O(n^2)$.

4. Estratégias de Robustez

O programa testa se o arquivo foi aberto corretamente, e se não, gera uma mensagem de erro. Além disso, no algoritmo de Graham, se for passado um código de tipo de método de ordenação inválido, diferente de 0, 1 ou 2, é impresso uma mensagem de erro, e é retornado uma pilha vazia. Tanto em Jarvis quanto em Graham, se houver menos de três pontos no vetor inicial ou no fecho formado, gera uma mensagem de erro na tela e é retornado um vetor vazio ou uma pilha vazia. Na função que calcula o produto escalar de dois vetores, há uma divisão. Se o denominador for 0, o produto escalar também é 0, então a função retorna 0.

5. Análise Experimental

Foram feitos testes com arrays de diferentes tamanhos para analisar a performance dos quatro tipos de algoritmo de fecho convexo:

Tempo	Graham Mergesort	Graham Insertion Sort	Graham Bucket Sort	Jarvis
10	0.000024s	0.000122s	0.000124s	0.000070s
50	0.000063s	0.000120s	0.000993s	0.000144s
100	0.000147s	0.000235s	0.003628s	0.000119s
200	0.000156s	0.000333s	0.028182s	0.000271s
500	0.000352s	0.000603s	0.118089s	0.000423s
1000	0.000813s	0.001267s	0.325667s	0.000858s

A partir desses testes, é possível perceber que os melhores métodos são o Jarvis e o Graham com Mergesort, já que o Graham com Mergesort possui complexidade O(n log n) sempre e o Jarvis dificilmente cai no pior

caso, então sempre fica com complexidade um pouco acima de O(n), que também é próximo de O(n log n).

Depois desses, vem o Graham com Insertion Sort, que tem complexidade entre O(n) e $O(n^2)$ e por isso varia dependendo da ordem inicial dos elementos do vetor, e por isso fica atrás dos dois algoritmos anteriores.

Por fim, temos o Bucket Sort, que é $O(n^2)$ sempre, e por isso tem uma performance pior do que todos os outros algoritmos sempre.

6. Conclusão

Para a execução desse Trabalho Prático, foi necessário entender como as estruturas de dados funcionam, para utilizá-las e adaptá-las para armazenar pontos e executar funções que permitissem que os algoritmos de ordenação e de obtenção do fecho convexo funcionassem bem.

Além disso, foi necessário entender o que é um fecho convexo, orientação de pontos, como calcular a distância entre pontos, ângulos entre pontos, produto escalar de vetores, além de como funcionam os métodos Scan de Graham e Marchar de Jarvis, para que pudesse haver a implementação do programa.

Por fim, percebe-se que é importantíssimo que haja a organização dos arquivos do programa em projetos que possuem grande quantidade de código, através da modularização e do uso do makefile. Além disso, fica nítida a importância do aprendizado acerca da utilização e funcionamento das estruturas de dados e dos métodos de ordenação, pois sem isso, seria praticamente impossível de obter uma solução para o problema apresentado. Saber qual estrutura utilizar e qual método de ordenação escolher para cada situação é muito importante para a performance do programa e para a facilidade da criação do código.

7. Bibliografia

- Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning, 2011.

- Cormen, T., Leiserson, C, Rivest R., Stein, C. Algoritmos – Teoria e Prática, 3a. Edição, Elsevier, 2012.

- GeeksForGeeks. 2023. Convex Hull using Graham Scan. Disponível em: www.geeksforgeeks.org/convex-hull-using-graham-scan/. Acesso em: 07 de jun. de 2023.

- GeeksForGeeks. 2022. Convex Hull using Jarvis' Algorithm or Wrapping. Disponível em: www.geeksforgeeks.org/convex-hull-using-jarvis-algorithm-or-wrapping/. Acesso em: 10 de jun. de 2023.

8. Instruções para compilação e execução

Para a compilação do programa, deve-se utilizar o comando “make all”, no terminal, estando na raiz da pasta TP.

Para a execução, deve se colocar o arquivo de teste na pasta TP, na raiz do projeto e digitar o comando estando na raiz do projeto, no terminal:

“./bin/fecho nome_arquivo.txt”, onde nome_arquivo.txt representa o nome do arquivo de testes.

Por exemplo, se o nome do arquivo for “testes.txt”, o comando deverá ser

```
./bin/fecho testes.txt
```