

TRABALHO PRÁTICO 2

Fecho Convexo

Universidade Federal de Minas Gerais - Departamento de Ciência da Computação
Estrutura de Dados - 2023/01 - Turma TN
Belo Horizonte - MG - Brasil

Nome: José Eduardo Duarte Massucato **Matrícula:** 2022043620
josedmass@ufmg.br

1. Introdução

O trabalho proposto tem por objetivo encontrar o melhor **fecho convexo**^[5] para um conjunto de pontos de entrada, cujas coordenadas são valores inteiros. Esse processo é feito a partir dos algoritmos *Scan de Graham*, que envolve métodos de ordenação, e *Marcha de Jarvis*, que serão explicitados posteriormente. Além disso, o programa retorna os tempos de execução para cada um dos algoritmos que, conforme descrito no enunciado do Trabalho Prático, são: *Scan de Graham* combinado com o *MergeSort*^[3], com o *InsertionSort*^[2] e com algum método de ordenação linear (foi escolhido o *BucketSort*^[4]), e a *Marcha de Jarvis*. Diante dessa perspectiva, o programa foi desenvolvido na linguagem C++, utilizando-se das estruturas de dados e classes apropriadas para lidar com os métodos citados.

2. Método

2.1. Estrutura de Dados

A estrutura de dados utilizada no programa foi a **pilha encadeada**. Ela foi escolhida para armazenar os pontos que formam o fecho convexo durante a execução do *Scan de Graham*, devido à natureza desse algoritmo, que envolve a comparação dos últimos pontos armazenados com o ponto atual a ser considerado. Dependendo do resultado dessa comparação, pode ser necessário remover o último ponto armazenado, correspondente ao topo da pilha. Diante dessa perspectiva, o uso da pilha facilita essa operação, pois, ao desempilharmos um elemento, estamos justamente removendo o seu topo.

2.2. Classes

2.2.1. class **Ponto**

Possui como **variáveis** `long long int x` e `long long int y`, que guardam, respectivamente, os valores das coordenadas `x` e `y` de um ponto passado como entrada. Possui como **funções**:

- `Ponto()`: constrói um `Ponto`. Note que esse construtor não faz nada, então é dever de quem cria uma variável do tipo `Ponto` atribuir valores para ele.
- `Ponto(long long int a, long long int b)`: constrói um `Ponto` com `x` igual a `a` e `y` igual a `b`.

2.2.2. class **Fecho**

Possui como **variáveis** `Ponto* fecho`, que armazena um conjunto de pontos no plano e `int tam`, que guarda o número atual de pontos armazenados no fecho.

Possui como **funções**:

- `Fecho()`: constrói um fecho vazio, com `tam` igual a 0 e `fecho` apontando para `NULL`.
- `Fecho& operator=(const Fecho& other)`: detalha o comportamento desejado para a atribuição de dois fechos. Ao final, o fecho `*this` fica com o mesmo tamanho e com os mesmos elementos que o fecho `other`.
- `bool operator==(const Fecho& other)`: detalha o comportamento desejado para a operação de igualdade de dois fechos. Caso o fecho `*this` tenha o mesmo tamanho e os mesmos elementos, dispostos na mesma ordem, que o fecho `other`, então essa função retorna `true`.
- `void Fecho Insere(Ponto p)`: insere um novo ponto ao fecho.
- `~Fecho()`: destrói o fecho, desalocando o espaço alocado em `fecho`.

2.2.3. class **PontoAngulo**

Possui como **variáveis** `Ponto p`, que armazena um ponto do plano e `double angulo`, que guarda o valor do ângulo, em radianos, que esse ponto faz em relação ao ponto inicial do fecho. Possui como **funções**:

- `PontoAngulo()`: constrói um `PontoAngulo`. Note que esse construtor não faz nada, então é dever de quem cria uma variável deste tipo atribuir valores para ela.

2.2.4. class **Celula**

Possui como **variáveis privadas** `PontoAngulo item`, que guarda um elemento do tipo `PontoAngulo`, e `Celula* prox`, que aponta para a próxima célula. Possui como **funções privadas**:

- `Celula()`: constrói uma célula vazia, isto é, `item` não recebe nada, por enquanto, e `prox` aponta para `NULL`.

2.2.5. class `PilhaEncadeada`

Possui como **variáveis privadas** `int tamanho`, que armazena o número atual de itens contidos na pilha, e `Celula* topo`, que aponta para o topo da pilha. Possui como **funções públicas**:

- `PilhaEncadeada()` : constrói uma pilha vazia, isto é, com tamanho igual a 0 e com topo apontando para `NULL`.
- `PilhaEncadeada& operator=(const PilhaEncadeada& other)` : descreve o comportamento desejado para atribuição de duas pilhas. Ao final, a pilha `*this` fica com o mesmo topo, mesmo tamanho e mesmos itens que a pilha `other`.
- `Celula* GetTopo()` é uma função auxiliar que retorna o topo da pilha, enquanto que `int GetTamanho()` é uma função auxiliar que retorna o tamanho atual da pilha. Ambas essas funções foram criadas com o intuito de garantir o encapsulamento do algoritmo.
- `PontoAngulo AbaixoDoTopo()` : função auxiliar que retorna o item da célula abaixo do topo da pilha. Essa função foi útil para a execução do *Scan de Graham*.
- `void Empilha_Item(PontoAngulo p)` : insere no topo da pilha um elemento do tipo `PontoAngulo`.
- `void Desempilha()` : deleta o atual topo da pilha, fazendo com que ele passe a ser igual à `topo->prox` e então diminuindo o tamanho da pilha em uma unidade.
- `Fecho ObterPlano()` : retorna o que está armazenado na pilha na forma do Tipo Abstrato de Dados `Fecho`.
- `void Limpa()` : limpa a pilha iterativamente e, ao final, faz com que topo aponte para `NULL`.
- `~PilhaEncadeada()` : destrói a pilha, chamando a função `Limpa()`.

2.3. Funções

Em `TP/src/main.cpp` leio, linha a linha, por meio do comando `std::getline`, o conteúdo do arquivo, até o seu fim. Então, passo essa linha para uma variável do tipo `std::istream`, para que eu consiga depois separar facilmente as coordenadas de um ponto, por meio dos espaços. Então, chamo a função `void exe_fecho(Fecho pl, std::string flag, double tempo[4])`, que executa os algoritmos que encontram o fecho convexo e mede o tempo de execução de cada um deles.

Observação: se `flag` for igual a “*graham+merge*”, então o *Scan de Graham* será executado utilizando o método *MergeSort* de ordenação; se for igual a “*graham+insert*”, então ele será executado com o método do *InsertionSort*; se for igual a “*graham+bucket*”, então ele será executado com o método do *BucketSort*; já se for igual a “*jarvis*”, então a Marcha de Jarvis será executada para encontrar o fecho convexo.

Observação: se no arquivo de entrada houver algum valor diferente de um número inteiro, então a exceção `ENTRADA_INVALIDA` é lançada.

2.3.1. Algoritmos de Ordenação

- `void merge(PontoAngulo vetor[], int inicio, int divisao, int fim):` é a parte da conquista do algoritmo do MergeSort, juntando ordenadamente, em relação ao ângulo de de cada item de `vetor`, os sub-vetores formados na fase de divisão.
- `void mergeSort(PontoAngulo vetor[], int inicio, int fim):` divide, recursivamente, o vetor de entrada, ordenando-o a partir da chamada da função `merge`, citada anteriormente. Aqui, temos que `inicio` é igual ao índice de onde começa o vetor e `fim` é o de onde ele termina.
- `void Insercao(PontoAngulo vetor[], int tam):` realiza o InsertionSort, ordenando `vetor` de tamanho `tam` em relação ao ângulo de cada um dos seus itens.
- `void bucketSort(PontoAngulo vetor[], int tam):` realiza o BucketSort, utilizando 9 baldes, ou seja, cada balde contém um intervalo de 20° dos ângulos (balde 1: $0 \leq \theta < 20$ / balde 2: $20 \leq \theta < 40$ / ... / balde 9: $160 \leq \theta < 180$). A partir disso, ele ordena cada balde individualmente utilizando o InsertionSort.

2.3.2. Funções Auxiliares

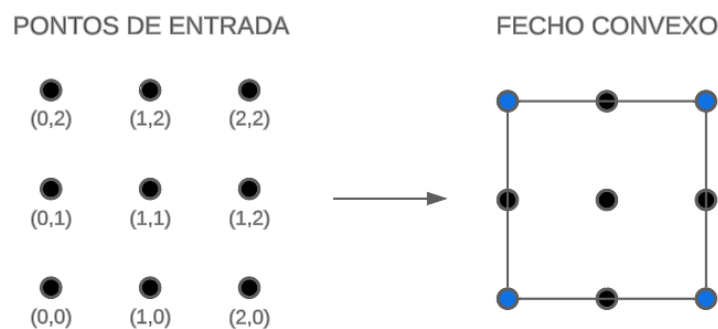
- `int IndicePontoMaisEmbaixoEsquerda(Fecho pontos):` retorna o índice de `pontos.fecho` do ponto com a menor coordenada y. Em caso de empate entre dois ou mais ângulos, ele escolhe o que tem a menor coordenada x.
- `int orientacao(Ponto p1, Ponto p2, Ponto p3):` calcula o produto vetorial^[6] entre esses três pontos. Por definição, se esse valor for positivo, significa que a reta formada pelos pontos `p1` e `p2` fazem uma rotação no sentido horário com a reta formada pelos pontos `p2` e `p3`, então a função retorna o valor 1. Se resultar em um valor negativo, então trata-se de uma rotação anti-horária e o valor retornado é 2. Caso o resultado seja igual a 0, então não houve rotação, ou seja, esses pontos são colineares.
- `long long int dist(Ponto p1, Ponto p2):` retorna a distância ao quadrado entre dois pontos. Essa função foi implementada para lidar com os casos de pontos colineares, como será explicitado posteriormente.
- `double calculaAngulo(Ponto p, Ponto central):` utiliza a função `std::atan2`, da biblioteca de matemática `<cmath>`, para calcular o ângulo formado entre os pontos `p` e `central`, em radianos.
- `PontoAngulo* VetorAngulos(Fecho pontos, Ponto central):` retorna um vetor de `PontoAngulo`, onde cada um dos seus elementos representa um dos pontos do arquivo de entrada e o ângulo que eles formam em relação com o ponto inicial (com menor coordenada y e, em caso de empate, com menor coordenada x do plano).

- `void printFecho(Fecho PontosFecho) :` imprime os pontos contidos no fecho convexo encontrado, de acordo com as especificações do Trabalho Prático.

2.3.3. Funções que encontram o Fecho Convexo

2.3.3.1. Fecho `MarchaDeJarvis(Fecho pontos)`

Essa função recebe o conjunto de pontos passados no arquivo de entrada, executa a Marcha de Jarvis e então retorna os pontos que formam o fecho convexo. A lógica dessa função é semelhante à explicitada no livro da disciplina^[4], a única alteração implementada foi o tratamento de casos onde a função `orientacao` retorna 0, isto é, caso haja pontos colineares. Nesses casos, a função seleciona apenas os pontos das extremidades. A imagem a seguir ilustra quais pontos ela escolhe (denotados em azul):



2.3.3.1. Fecho `ScanDeGraham(Fecho pontos, std::string sort)`

Essa função recebe o conjunto de pontos passados no arquivo de entrada, executa o *Scan de Graham* com a ordenação definida pelo parâmetro `sort` e então retorna os pontos que formam o fecho convexo. A lógica por trás dessa função se baseia na explicitada no livro da disciplina, mas com algumas alterações. Após a ordenação, empilhamos, inicialmente, apenas dois itens na pilha, ao invés de três como é no livro, para facilitar no tratamento do caso de pontos colineares. Assim como na Marcha de Jarvis, essa função também seleciona apenas os pontos das extremidades em casos de pontos colineares.

Observação: se `sort` for igual à “merge”, então a ordenação utilizada no *Scan de Graham* será o *MergeSort*; se for igual à “insert”, então será o *InsertionSort*; já se for igual à “bucket”, então será o *BucketSort*.

3. Análise de Complexidade

Primeiramente, vale destacar a complexidade de tempo e espaço das funções citadas na **seção 2.3**:

Função	Complexidade de tempo	Complexidade de Espaço
IndicePontoMaisEmbaixoEsquerda	$O(n)$	$O(1)$
orientacao	$O(1)$	$O(1)$
dist	$O(1)$	$O(1)$
VetorAngulos	$O(n)$	$O(n)$
Pilha::ObterFecho	$O(n)$	$O(n)$
Fecho::Insere	$O(n)$	$O(n)$
Insercao	$O(n)$, no melhor caso	$O(1)$
	$O(n^2)$, no pior caso	
MergeSort	$O(n \log n)$	$O(n)$
BucketSort	$O(n^2/9) = O(n^2)$	$O(1)$

3.1. Scan de Graham

Chama `IndicePontoMaisEmbaixoEsquerda`, depois chama `VetorAngulos` e então utiliza algum dos métodos de ordenação. Depois disso, faz uma iteração cuja complexidade de tempo é, no pior caso, $O(n^2)$ e no melhor caso é $O(n)$, a depender do número de vezes que ele desempilha, uma vez que ele chama as funções `orientacao` e `dist`. Por fim, ele chama `Pilha::ObterFecho`. Diante disso, temos a seguinte **complexidade de tempo**:

- Se o método de ordenação utilizado for o MergeSort, então a complexidade de tempo é dada por: $O(n) + O(n) + O(n \log n) + \text{iteração}$, ou seja, ele é $O(n \log n)$ no melhor caso e $O(n^2)$ no pior caso.
- Se o método de ordenação utilizado for o InsertionSort, então a complexidade de tempo é dada por: $O(n) + O(n) + \text{sort} + \text{iteração}$, ou seja, ele é $O(n)$ no melhor caso e $O(n^2)$ no pior caso.
- Se o método de ordenação utilizado for o BucketSort, então a complexidade de tempo é dada por: $O(n) + O(n) + O(n^2) + \text{iteração}$, ou seja, ele é sempre $O(n^2)$.

Para todos esses casos, a **complexidade de espaço** é delimitada por $O(n)$.

3.2. Marcha de Jarvis

Chama a função `IndicePontoMaisEmbaixoEsquerda`, depois faz uma iteração cuja complexidade de tempo é dada por $O(n^2)$ e de espaço é dada por $O(n)$, uma vez que ela chama as funções `Fecho::Insere`, `orientacao` e `dist`. Diante dessa perspectiva, temos a seguinte complexidade de tempo total:

$$O(n) + n(O(n)) = O(n) + O(n^2) = O(\max(n, n^2)) = O(n^2)$$

Em relação a complexidade de espaço, temos que ela é delimitada por $O(n)$.

4. Estratégias de Robustez

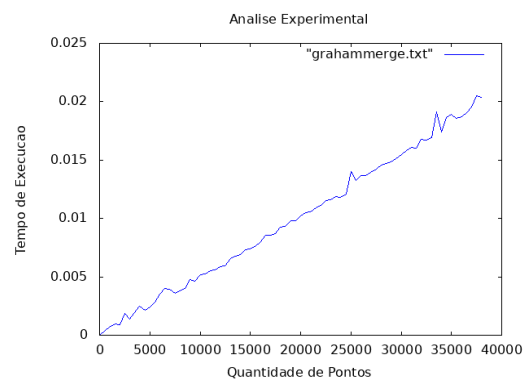
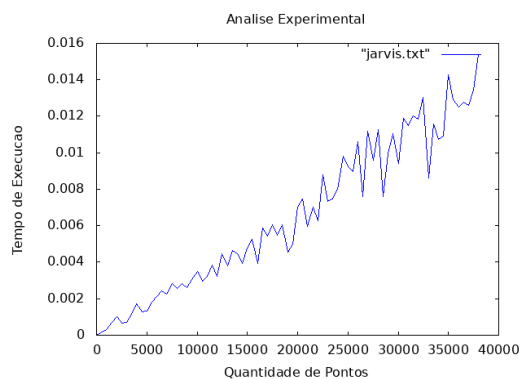
A fim de tornar o programa mais robusto, um tratamento de exceção foi implementado, contido em *TP/include/Excecoes.h*:

- `ENTRADA_INVALIDA`: É lançada caso no arquivo de entrada haja um valor que não é um número inteiro, isto é, caso haja letras, números com vírgula, símbolos, entre outros. Uma mensagem de erro alertando o problema é então exibida na tela.

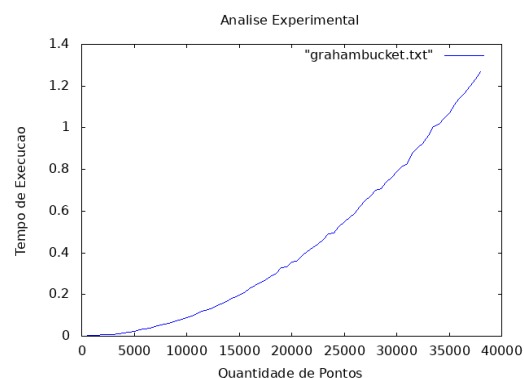
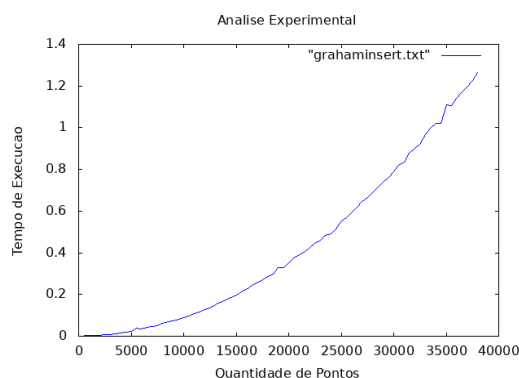
Caso o usuário não coloque o nome do arquivo no momento da compilação, como será descrito na **seção 8**, uma mensagem de erro é exibida na saída padrão. Da mesma forma, se o arquivo não abriu adequadamente, uma mensagem de erro é exibida.

5. Análise Experimental

A análise do tempo de execução do programa foi feita com base na quantidade de pontos inseridos no arquivo de entrada. Quatro gráficos foram construídos, cada um representando o algoritmo utilizado para encontrar o fecho convexo. A análise foi feita para entradas no intervalo de 0 a 38000 pontos.



Os dois gráficos acima representam, respectivamente, os algoritmos da Marcha de Jarvis e do Scan de Graham com o método MergeSort de ordenação. Como discutido na **seção 3**, ambos têm complexidade de tempo delimitada por $O(n^2)$. Note que a Marcha de Jarvis é bastante sensível à entrada, pois, a depender da maneira na qual um ponto é inserido na estrutura Fecho, o desempenho desse algoritmo pode variar. Além disso, percebe-se que o Scan de Graham, quando aliado ao MergeSort, parece ter um gasto de tempo linear. Isso ocorre pelo fato de esse algoritmo ser $O(n \log n)$ no caso médio.



Já os gráficos acima representam o algoritmo do Scan de Graham com os métodos do InsertionSort e do BucketSort, respectivamente. Como discutido na **seção 3**, ambos têm complexidade de tempo determinada por $O(n^2)$. Note que o gráfico do BucketSort se assemelha muito ao do InsertionSort, muito pelo fato de que, como comentado na **seção 2.3.1**, ele realiza o InsertionSort em cada um dos seus baldes.

6. Conclusões

Neste trabalho, foram desenvolvidas diversas classes e funções para lidar com os pontos em um plano e para formar um fecho convexo com eles. Ao longo do desenvolvimento deste programa, a principal estrutura de dados desenvolvida e adaptada ao problema foi a pilha encadeada, fato que levou ao aprendizado sobretudo da alocação dinâmica, imprescindível para o funcionamento dela. Além disso, a utilização dos algoritmos de ordenação foi indispensável para o aprendizado dos mesmos, uma vez que foi preciso adaptá-los ao problema do Trabalho Prático. Por outro lado, elementos fundamentais da computação, como análise de complexidade de algoritmos e implementação de estratégias de robustez, foram desenvolvidos no decorrer do trabalho. Em suma, esse trabalho estava adequadamente alinhado ao que estávamos aprendendo na disciplina de Estrutura de Dados e garantiu uma boa curva de aprendizado.

7. Bibliografia

1. CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Introduction to Algorithms. 3rd ed. Cambridge: MIT Press, 2009. Seção 33: Computacional Geometry;
2. Insertion Sort. Disponível em: <<https://www.programiz.com/dsa/insertion-sort>>. Acesso em: 2 jun. 2023;
3. Merge Sort. Disponível em : <<https://www.programiz.com/dsa/merge-sort>>. Acesso em: 2 jun. 2023;
4. Bucket Sort. Disponível em : <<https://www.programiz.com/dsa/bucket-sort>>. Acesso em: 2 jun. 2023;
5. Convex Hull. In: Encyclopedia of Mathematics. Disponível em: <https://encyclopediaofmath.org/wiki/Convex_hull> . Acesso em: 8 jun. 2023;
6. Cross Product. In: Encyclopedia of Mathematics. Disponível em: <https://encyclopediaofmath.org/wiki/Cross_product>. Acesso em: 8 jun. 2023.

8. Instruções para compilação e execução

8.1. Abra o terminal.

8.2. Entre na pasta TP

- **Comando:** `cd / <caminho para a pasta > /TP`

8.3. Adicione o(s) arquivo(s) de entrada em ./TP

8.4. Compile e execute o programa a partir do arquivo Makefile presente em ./TP

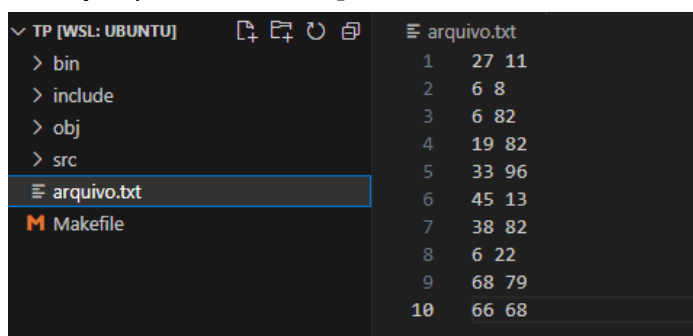
- **Comando:** `make fecho <file_name.txt>`
- *Observação 1:* este comando compila e executa o programa;
- *Observação 2:* o arquivo **deve** estar na pasta TP
- *Observação 3:* a cada nova execução, `<file_name.txt>` deve ser substituído pelo nome do arquivo que se deseja operar, com extensão `.txt`.

8.5. Como especificado no enunciado do TP, o resultado da execução do programa é imprimido na **saída padrão**.

8.6. Ao final de todas as execuções, utilizar o Makefile para limpar os arquivos gerados

- **Comando:** `make clean`

Exemplo para ./TP/arquivo.txt



TP [WSL: UBUNTU]	arquivo.txt
> bin	1 27 11
> include	2 6 8
> obj	3 6 82
> src	4 19 82
≡ arquivo.txt	5 33 96
M Makefile	6 45 13
	7 38 82
	8 6 22
	9 68 79
	10 66 68

```
root@DESKTOP-01SI1LT:~/home/josedmass/ED/TP2/TP# make fecho arquivo.txt
g++ -Wall -c -Iinclude -o obj/main.o src/main.cpp
g++ -Wall -c -Iinclude -o obj/FechoConvexo.o src/FechoConvexo.cpp
g++ -Wall -c -Iinclude -o obj/Ponto-e-Fecho.o src/Ponto-e-Fecho.cpp
g++ -Wall -c -Iinclude -o obj/Sorts.o src/Sorts.cpp
g++ -Wall -c -Iinclude -o obj/PilhaEncadeada.o src/PilhaEncadeada.cpp
g++ -pg -o bin/main obj/main.o obj/FechoConvexo.o obj/Ponto-e-Fecho.o obj/Sorts.o obj/PilhaEncadeada.o -lm
./bin/main arquivo.txt
FECHO CONVEXO:
6 8
45 13
66 68
68 79
33 96
6 82

GRAHAM+MERGESORT: 0.177s
GRAHAM+INSERTIONSORT: 0.000s
GRAHAM+LINEAR: 0.000s
JARVIS: 0.000s
make: Nothing to be done for 'arquivo.txt'.
```

```
root@DESKTOP-01SI1LT:~/home/josedmass/ED/TP2/TP# make clean
rm -f bin/main obj/main.o obj/FechoConvexo.o obj/Ponto-e-Fecho.o obj/Sorts.o obj/PilhaEncadeada.o gmon.out
```