Trabalho Prático 02 - Fecho Convexo

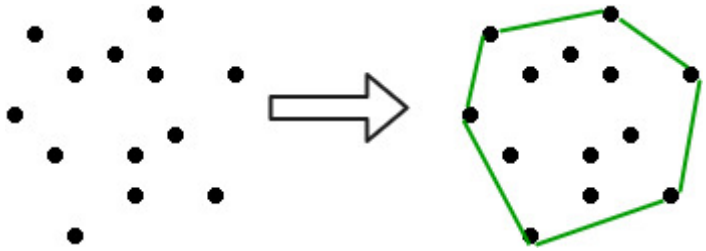
Aluno: Raphael Henrique Braga Leivas
Matrícula: 2020028101

Universidade Federal de Minas Gerais - UFMG
Belo Horizonte

1 INTRODUÇÃO

O Trabalho Prático 02 consiste essencialmente no desenvolvimento de um identificador do fecho convexo de um conjunto de pontos. O fecho convexo é definido como o menor polígono convexo que engloba um conjunto de pontos, como mostrado na Figura 1.1.

Figura 1.1. Fecho convexo (destacado em verde) de um conjunto de pontos.



Fonte: Disponível em <https://iq.opengenus.org/gift-wrap-jarvis-march-algorithm-convex-hull/>. Acesso em 16 de jun. de 2023.

Os pontos de entrada são dados em um arquivo de texto, que será lido pelo software e os pontos que compõem o fecho convexo serão exibidos na saída padrão (std::cout). Além disso, será exibido opcionalmente as equações das retas que compõem o fecho na forma $y = ax + b$.

Tendo em vista a aplicação de Estruturas de Dados e algoritmos vistos na disciplina, será usado a linguagem C++ para desenvolver três classes, representando três Tipos Abstratos de Dados (TADs) Ponto, Reta e Fecho Convexo. Além disso, usaremos estruturas auxiliares, tais como Lista Encadeada, Lista Arranjo e Pilha Arranjo, desenvolvidas de autoria própria, com o auxílio dos slides da disciplina.

Para a determinação do fecho serão usados os algoritmos prontos de geometria computacional da marcha de Jarvis e do scan de Graham, disponíveis tanto no livro-texto de referência da disciplina quanto em diversos sites da internet, adaptados para os TADs e estruturas usadas.

Para aplicar os algoritmos de ordenação vistos em sala, o scan de Graham será feito com três algoritmos diferentes de ordenação:

- Mergesort
- Insertion sort
- Counting sort

Por fim, será feita uma análise e comparação entre os desempenhos dos diferentes algoritmos de ordenação, usando ferramentas também vistas durante a disciplina.

Dessa forma, o TP será capaz de reforçar conceitos da disciplina vistos até agora e aprimorar o nosso conhecimento com estruturas já conhecidas, bem como ter um contato com algoritmos de geometria computacional e ordenação.

2 MÉTODO

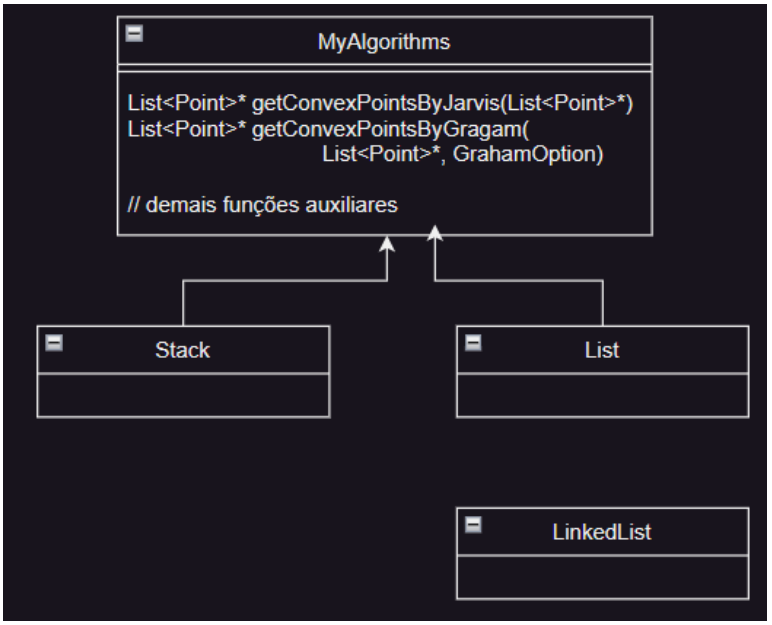
O software foi desenvolvido em linguagem C++ em ambiente Windows, sendo compilado em WSL Ubuntu 22.04 LTS com g++. O processo de compilação dos módulos é gerido por Makefile, assim mantendo a estrutura da pasta especificada em atividades da disciplina. Para verificar o gerenciamento de memória da aplicação é usado o valgrind, ao passo que para depurar a aplicação foi usado o gdb.

A análise em tempo de execução foi feita usando o gprof, com os dados da análise sendo escritos em arquivo txt após a compilação e execução do código. Uma biblioteca de autoria própria customTime.h é usada para computar o tempo gasto pelas funções usando Unix Timestamp.

2.1 ESTRUTURAS DE DADOS

A Figura 2.1 mostra as principais estruturas de dados usadas que efetuam os algoritmos do software.

Figura 2.1. Estruturas de dados da solução.



Fonte: elaboração própria.

A classe MyAlgorithms é essencialmente um conjunto de funções que usamos durante o programa. A grande maioria desses métodos são privados, de modo que quem chama essa classe (no caso, o programa principal) apenas enxerga dois métodos:

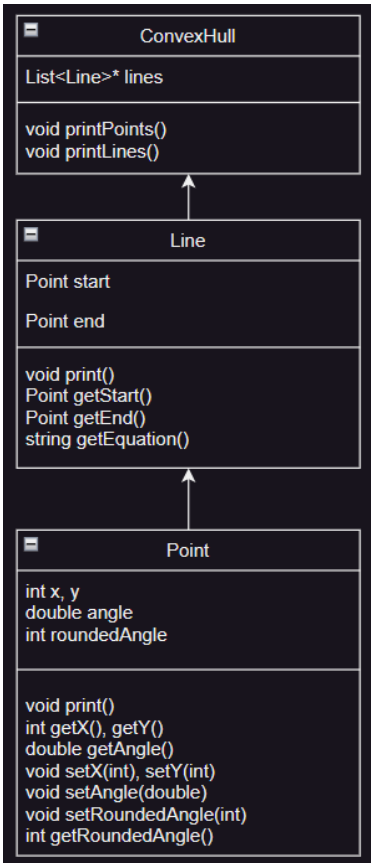
- getConvexPointsByJarvis(), que recebe uma lista de pontos que se deseja identificar o fecho convexo e retorna uma outra lista de pontos, que representa os pontos que compõem o fecho convexo via marchar de Jarvis;
- getConvexPointsByGraham(), que tem a mesma interface e função que o método acima, só que desta usando o scan de Graham. Esse método também recebe um segundo argumento, uma enumeração que indica se deve-se usar como algoritmo de ordenação o mergeSort, countingSort ou o insertionSort.

A classe MyAlgorithms chama as estruturas Lista Arranjo (List) e Pilha Arranjo (Stack), que são usadas durante os algoritmos e cálculos. Note que a estrutura Lista Encadeada (LinkedList) não é chamada pelos algoritmos, uma vez que ela é usada apenas pelo programa principal para ler o arquivo de entrada, já que não sabemos qual o tamanho do arquivo de entrada a priori. Dessa forma, o programa lê a entrada, salva os pontos na Lista Encadeada, converte a Lista Encadeada em uma Lista Arranjo (uma vez que agora sabemos quantos pontos estamos trabalhando, logo podemos usar array), e depois trabalha com os algoritmos apenas com a List Arranjo.

De fato, seria melhor que os algoritmos usassem a Lista Encadeada de uma vez em vez de converter para Lista Arranjo, mas isso foi feito para que possamos trabalhar menos com ponteiros e assim ter menos erros de manipulação de memória (segmentation faults).

A Figura 2.2 exibe os 3 TADs que representam geometria no software, bem como a relação entre eles. Esses TADs usam as estruturas desenvolvidas discutidas na Figura 2.1.

Figura 2.2. TADs de geometria do software.



Fonte: elaboração própria.

Note, na Figura 2.2, que uma Reta é um conjunto dois de Pontos, e um Fecho Convexo é um conjunto de Retas, na qual representamos como uma Lista de Retas.

O TAD Point possui dois atributos auxiliares: angle e roundedAngle. Ambos representam o ângulo polar em relação a um ângulo de referência, e são usado no Scan de Graham. RoundedAngle é exatamente o mesmo valor de angle mas arredondado para a terceira casa decimal e multiplicado por 1000, de modo que ele vira um número inteiro. Isso é feito apenas para a ordenação do countingSort, que só trabalha com inteiros. As ordenações mergeSort e inserionSort usam o angle do tipo double normalmente.

2.2 IMPLEMENTAÇÃO DAS ESTRUTURAS

As classes Stack e Lista implementam o TAD Pilha e Lista na forma de arranjos e vetores, isto é, com alocação estática, de maneira semelhante às vistas na disciplina. A classe LinkedList implementa o TAD Lista Encadeada também de forma bastante parecida com os slides da disciplina. Assim, vamos focar na implementação dos dois métodos principais da classe MyAlgorithms.

2.2.1 Cálculo do fecho convexo via Marchar de Jarvis

Implementado no método getConvexPointsByJarvis(). Recebe uma Lista de Pontos e retorna uma outra Lista de Pontos, que representa o fecho convexo dos pontos de entrada. A implementação desse método se baseou fortemente no artigo online “Convex Hull using Jarvis’ Algorithms or Wrapping”, disponível nas referências dessa documentação.

Essencialmente, a função implementada efetua o seguinte passo a passo:

1. Busca na Lista de Pontos de entrada aquele ponto que tem a menor coordenada x, ou seja, aquele mais à esquerda do conjunto;
2. A partir desse ponto, comece a percorrer todos os pontos em sentido anti-horário, ou seja, indo para o ponto que está com orientação mais anti-horária em relação ao ponto atual. Isso é determinado pela função orientation, privada na classe MyAlgorithms.
3. Salva o ponto atual que está percorrendo na Lista de Pontos do fecho convexo
4. Repita 2 e 3 até chegar no ponto inicial, feito em um loop do-while.

Feito isso, basta retornar a Lista de Pontos final ao programa principal.

2.2.2 Cálculo do fecho convexo via Scan de Graham

Implementado no método getConvexPointsByGraham(). A implementação desse método se baseou fortemente em duas referências: o pseudocódigo no livro-texto do Cormen, exibido na Figura 2.3, e o artigo “Convex Hull using Graham Scan”, também disponível nas referências.

Figura 2.3. Pseudocódigo do Scan de Graham usado como referência.

```
GRAHAM-SCAN(Q)
 1 seja p_0 o ponto em Q com a coordenada y mínima
       ou tal ponto que esteja mais à esquerda no caso de empate
 2 sejam ⟨p_1, p_2, ..., p_m⟩ os pontos restantes em Q,
         ordenados por ângulo polar em ordem anti-horária em torno de p_0
         (se mais de um ponto tiver o mesmo ângulo, remover todos eles, exceto o mais
         afastado de p_0)
 3 seja S uma pilha vazia
 4 PUSH(p_0, S)
 5 PUSH(p_1, S)
 6 if m ≥ 2 PUSH(p_2, S)
 7 for i = 3 to m
 8   while o ângulo formado pelos pontos NEXT-TO-TOP(S),
               TOP(S) e p_i curva não vira para a esquerda
 9         POP(S)
10       PUSH(p_i, S)
11 return S
```

Fonte: CORMEN, Thomas H., et al. (2012).

Olhando o código-fonte desse método, teremos essencialmente uma mistura entre os passos do Cormen e do artigo, pelos comentários é possível identificar qual passo está sendo executado.

Uma diferença fundamental nesse método está em uma condicional switch-case, que identifica qual algoritmo de ordenação deve ser usado a partir da opção fornecida no segundo argumento do método. Dentro do switch-case, temos três funções auxiliares que podem ser chamadas dependendo da opção fornecida:

- sortByAngleInsertionSort(): ordena o vetor de pontos a partir do ângulo polar entre o ponto de referência e o ponto atual via insertionSort;
- sortByAngleMergeSort(): mesma função que o método acima, usando mergeSort;
- sortByAngleCountingSort(): mesma função que o método acima, usando countingSort.

A implementação desses algoritmos de ordenação se baseou bastante nos slides da disciplina e algumas referências online. São essencialmente os mesmos algoritmos que os vistos na disciplina, adaptados ao TAD Point, o qual possui o atributo angle. Assim, os pontos são ordenados com base nesse atributo angle (ou roundedAngle, para o coutingSort), que fica na classe Point.

No final do método, é retornada uma Lista de Pontos ao programa principal, representando os pontos do fecho convexo calculado.

3 ANÁLISE DE COMPLEXIDADE

3.1 MARCHA DE JARVIS

Observando o código fonte do método getConvexPointsByJarvis(), temos um loop for sozinho, seguido de um loop do-while com um outro loop for nestado. O loop for nestado (interno) pode ser iterado n vezes, onde n é o número de pontos de entrada, ao

passo que o loop do-while externo pode ser interado h vezes, onde h é o número de pontos pertencentes ao fecho convexo. No pior caso em que $h = n$, temos complexidade assintótica $O\left(n^2\right)$.

3.2 SCAN DE GRAHAM

O Scan de Graham pode ser dividido em três partes principais:

1. Encontrar o ponto com menor coordenada y. Basta percorrer o vetor de entrada uma vez até achar o ponto desejado, no qual obtemos em $O(n)$.
2. Ordenar o vetor pelo ângulo polar.
    a. Caso seja usado insertionSort, temos que é gasto $O\left(n^2\right)$, uma vez que o pior caso temos dois loops nestados percorridos duas vezes (vetor ordenado com ângulos polares em forma decrescente).
    b. Caso seja usado mergeSort, temos $O(n\ log(n))$, pois é gasto $O(log\ n)$ para dividir o array e $O(n)$ para juntar.
    c. Por fim, caso seja usado coutingSort, gasta-se $O(n + k)$, onde k é a faixa de valores do vetor. Os valores do ângulo polar podem variar de 0 a 180° graus, ou seja, de 0 a 3.141 radianos. Como o countingSort trabalha com inteiros (índices i dos vetores são inteiros), convertemos os radianos para inteiro multiplicando por 1000 no código, mantendo precisão de 3 casas decimais conforme especificado nos requisitos do TP. Assim, o coutingSort tem complexidade linear.
3. Construir o fecho convexo usando a Pilha. Note, no código-fonte, que temos um loop for e dentro dele um loop while, o que sugere que essa etapa é $O\left(n^2\right)$. Contudo, como cada elemento é empilhado e desempilhar no máximo 1 vez, e as operações de empilhar e desempilhar tem complexidade $O(1)$, temos que essa etapa tem complexidade $O(n)$.

Assim, a complexidade do Scan de Graham depende essencialmente de qual algoritmo vamos usar, uma vez que os passos 1 e 3 são $O(n)$ e o passo 2 tem complexidade sempre maior ou igual a $O(n)$. A Tabela 3.1 sumariza a análise que fizemos nessa seção.

Tabela 3.1. Resumo da análise de complexidade feita nessa seção.

Algoritmo	**Complexidade**
Jarvis	$O\left(n^2\right)$
Graham + insertionSort	$O\left(n^2\right)$
Graham + mergeSort	$O(n\ log(n))$
Graham + countingSort	$O(n + k)$

Fonte: elaboração própria.

4 ESTRATÉGIAS DE ROBUSTEZ

No programa principal, o arquivo de entrada é lido e salva os pontos em uma Lista Encadeada. Concluído a leitura, é feita uma verificação do tamanho dessa lista: se o tamanho for zero, temos que o arquivo de entrada está vazio (ou ele não existe). Nesse cenário, o código emite uma mensagem na saída padrão e retorna do programa principal, sem chamar os algoritmos.

Além disso, as funções dos algoritmos retornam uma mensagem de erro (exceção) se chegar menos que 3 pontos na lista de pontos de entrada, uma vez que não é possível, teoricamente, extrair o fecho convexo de um conjunto com menos de três pontos.

5 ANÁLISE EXPERIMENTAL

A Tabela 5.1 exibe os tempos gastos por cada algoritmo em microsegundos para entradas de tamanho 10, 100 e 1000 pontos (são as entradas de exemplo disponibilizadas no moodle).

Tabela 5.1 Tempos de execução para cada um dos algoritmos de ordenação do scan de Graham.

Algoritmo de ordenação	Tempo gasto (μs)		
	Entrada 10	Entrada 100	Entrada 1000
mergeSort	112	239	1953
insertionSort	11	302	6289
countingSort	104	220	1479
Jarvis	13	196	3463

Fonte: elaboração própria.

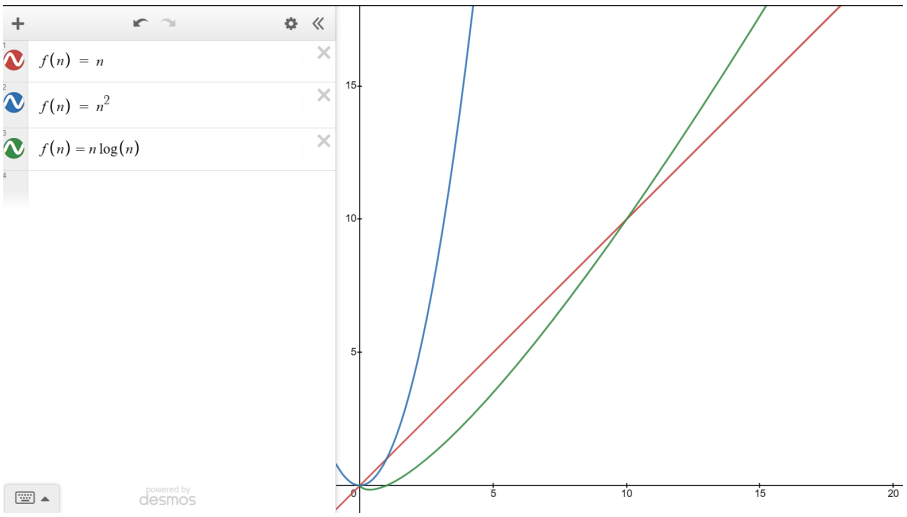
É bastante interessante observar na Tabela 5.1 que os tempos gastos na execução estão diretamente relacionados à complexidade assintótica dos algoritmos de ordenação, feita na Tabela 3.1. Conforme o tamanho do arquivo de entrada aumenta, o insertionSort é o que gasta mais tempo, uma vez que tem complexidade $O\left(n^2\right)$. De fato, comparando os tempos gastos entre as entradas do insertionSort, vemos um aumento quadrático dos tempos de execução.

O mergeSort, por outro lado, possui complexidade $O(n\,log(n))$, apresentando um crescimento mais lento que o insertionSort. Por fim, o countingSort apresenta complexidade linear $O(n + k)$ como discutido na seção 3, sendo por consequência o mais rápido dos três algoritmos. Como vemos na Tabela 5.1, os tempos gastos entre as execuções apresentam crescimento linear conforme o aumento da entrada. Note que ele é um pouco mais rápido que o mergeSort, mas é extremamente mais rápido que o insertionSort.

Por fim, analisamos também o Jarvis que possui complexidade $O(n^2)$ e, assim como o Graham + insertionSort, tem tempos de execução que crescem muito rápido com o aumento da entrada.

A Figura 5.1 mostra o crescimento das funções de complexidade assintótica dos três algoritmos, de modo a visualizar melhor o crescimento. Note que a curvas refletem (em parte) o comportamento visto nos dados da Tabela 5.1.

Figura 5.1. Crescimento das funções de complexidade assintótica do coutingSort (vermelho), mergeSort (verde) e insertionSort / Jarvis (azul).



Fonte: elaboração própria.

6 CONCLUSÕES

Tendo em vista os objetivos do TP, foi possível desenvolver um sistema que calcula o fecho convexo a partir de um conjunto de pontos de entrada. Durante o desenvolvimento, foi possível aprender sobre algoritmos de geometria computacional e algoritmos de ordenação integrados à TADs específicos da aplicação. Além disso, TADs já vistos na primeira parte da disciplina - Lista e Pilhas - também foram abordados em maior profundidade no software.

Por fim, foi bastante interessante perceber na prática o impacto que a complexidade assintótica tem nos tempos gastos de execução conforme o aumento do arquivo de entrada.

7 REFERÊNCIAS

CORMEN, Thomas H., et al. Algoritmos. Teoria e Prática. 3 ed. Rio de Janeiro. Elsevier Editora. 2012.

CHAIMOWICZ, Luiz. PRATES, Raquel. Slides Estruturas de Dados: Pilhas e Filas. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais.

CHAIMOWICZ, Luiz. PRATES, Raquel. Slides Estruturas de Dados: Análise de Complexidade e Complexidade Assintótica. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais.

CHAIMOWICZ, Luiz. PRATES, Raquel. Slides Estruturas de Dados: Métodos de ordenação sem comparação de chaves. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais.

CHAIMOWICZ, Luiz. PRATES, Raquel. Slides Estruturas de Dados: Ordenação: MergeSort. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais.

CHAIMOWICZ, Luiz. PRATES, Raquel. Slides Estruturas de Dados: Ordenação: Métodos Simples. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais.

GEEKS FOR GEEKS. Convex Hull using Jarvis' Algorithms or Wrapping. Disponível em: <https://www.geeksforgeeks.org/convex-hull-using-jarvis-algorithm-or-wrapping/>. Acesso em 09 de jun. 2023.

GEEKS FOR GEEKS. Convex Hull using Graham Scan. Disponível em: <https://www.geeksforgeeks.org/convex-hull-using-graham-scan/>. Acesso em 10 de jun. 2023.

SHARMA, Pankaj. Gift warp algorithm (Jarvis March algorithm) to find convex hull. Disponível em <https://iq.opengenus.org/gift-wrap-jarvis-march-algorithm-convex-hull/>. Acesso em 16 de jun. de 2023.

8 INSTRUÇÕES DE COMPILAÇÃO E EXECUÇÃO

Siga o passo a passo abaixo para compilar e executar o TP:

1. Tenha um sistema Linux com compilador g++ e make instalado e configurado;
2. Rode o comando `make clean`
3. Rode o comando `make run`
    a. Observação: após o make run, será printado na saída que o arquivo de entrada não existe. Para passar o arquivo de entrada, use o passo 4 abaixo.
4. Após o make run, dentro da pasta bin estará o arquivo binário “fecho” que pode ser executado. Rode o comando `./bin/fecho FILE_PATH`, onde FILE_PATH é o caminho completo do arquivo txt que os pontos de entrada, no mesmo formato que o arquivo de exemplo disponibilizado no moodle. Exemplo:

./bin/fecho /mnt/c/dev/estruturas-de-dados-2023-1/TP02/exemplosTP2/ENTRADA10.txt

Caso deseje ver as equações das retas do fecho convexo calculado, use a flag -r no arquivo binário antes do FILE_PATH:

./bin/fecho -r /mnt/c/dev/estruturas-de-dados-2023-1/TP02/exemplosTP2/ENTRADA10.txt