Trabalho Prático 2
Fecho Convexo

Rafael Martins Gomes - 2022043779

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte - MG - Brasil

rafaelmgomes.dev@gmail.com

1. Introdução

A documentação a seguir diz respeito à criação e implementação de um sistema que, dado um conjunto de pontos, determina o fecho convexo. O problema foi inspirado numa situação fictícia, na qual um funcionário de uma empresa da indústria têxtil deve auxiliar seu chefe a determinar os melhores tamanhos de peças de um tecido extremamente caro e apenas vendido em pedaços convexos. Para isso, o funcionário deve encontrar o menor polígono convexo que encapsule todos os pontos passados por seu chefe.

O programa faz uso dos algoritmos “scan de Graham” e “marchar de Jarvis” para solucionar o problema. Para o scan de Graham, foram implementados 3 algoritmos de ordenação.

2. Método

O trabalho foi desenvolvido em uma máquina com as seguintes especificações:
Sistema operacional: Windows 11/WSL - Ubuntu 20.04 LTS
Linguagem de programação: C++
Compilador: g++ 9.4.0
Processador: AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz
Memória RAM: 16,0 GB

2.1. Estruturas de dados

Para armazenar o conjunto de pontos a ser analisado, foi utilizado um vetor de pontos. Esse vetor, assim como o vetor que armazena os pontos que formam o fecho convexo, faz parte de uma classe que gerencia e constrói o fecho convexo. Um ponto é representado por uma classe que tem como atributos as coordenadas X e Y, armazenadas como inteiros.

Outros vetores auxiliares também foram utilizados ao longo do programa pelas funções de ordenação e pelos algoritmos que constroem o fecho. Uma pilha também foi utilizada pelo scan de Graham.

2.2. Classes

O programa contém 4 classes. As classes “Stack” e “Item” dizem respeito à implementação de uma pilha e de seus elementos. A pilha foi implementada de modo que seus itens contenham dados de tipos que serão especificados apenas na instanciação de um objeto dessa classe. No programa em questão, ela é instanciada como uma pilha de pontos.

Nesse sentido, um ponto é representado pela classe “Point” que, como colocado anteriormente, contém inteiros que armazenam suas coordenadas X e Y. Ela também permite acesso a seus atributos privados por funções auxiliares utilizados para ordenar os pontos e construir o fecho - para isso, declara tais funções como *friends*.

Por fim, a classe “ConvexHull” contém, além de um vetor para o conjunto de todos os pontos e um para aqueles que formam o fecho convexo, como colocado acima, o número de elementos de cada um desses vetores, as funções de ordenação dos pontos e os algoritmos de construção do fecho. É importante ressaltar que, apesar de ser utilizada para construir o fecho, a função do Merge Sort não faz parte da classe do fecho convexo, visto que também é usada na função de ordenação linear (que será destrinchada mais a frente) para ordenar conjuntos de pontos diferentes.

2.3. Funções/métodos

O código tem como principais funções e métodos os seguintes: *ccw*, *comparePoints*, *mergeSort*, ConvexHull::*insertionSort*, ConvexHull::*bucketSort*, ConvexHull::*grahamScan* e ConvexHull::*jarvisMarch*. Outras funções auxiliares também são usadas na execução dessas.

As funções *ccw* e *comparePoints* são usadas para determinar a orientação de um conjunto de três pontos e, a partir dessa, determinar qual deve ser colocado antes ao se ordenar o vetor de pontos. Na função ccw, calcula-se a inclinação dos segmentos de reta formados por *point1* e *point2* e por *point2* e *point3*. Caso o segundo segmento de reta tenha uma inclinação menor que o primeiro, o sentido de rotação dos pontos é horário; caso a inclinação dos dois seja igual, eles estão na mesma linha; se o segundo segmento tiver inclinação maior que o primeiro então os pontos estão no sentido anti-horário. Como a ideia é ordenar os pontos em ordem crescente seguindo esse último caso, a função *comparePoints* é usada para determinar qual dos pontos deve ser considerado o “menor” pelas funções de ordenação a partir desse critério.

O método *mergeSort* implementa o método de ordenação de mesmo nome. Nesse algoritmo, divide-se o vetor a ser ordenado em sub-vetores de 1 elemento cada, e compara-se cada sub-vetor com o vetor adjacente para juntá-los em um

novo sub-vetor ordenado. Repete-se esse processo até que haja apenas um vetor, que conterá os pontos ordenados. A função *merge* é utilizada para realizar a junção entre dois sub-vetores e, nela, usa-se a função *comparePoints* para comparar os ângulos formados pelo ponto mais baixo do conjunto e os outros pontos.

O método *insertionSort*, por sua vez, funciona da seguinte forma: passa-se por todos os pontos do vetor e compara-os com os pontos que estão antes dele (usando, novamente, a função *comparePoints* e o sentido anti-horário como critério de ordenação). Se os antecessores de um ponto tem o ângulo maior que o dele, move-os para frente no vetor, de modo a “abrir espaço” para inserção do ponto com ângulo menor na posição correta. Depois desse processo ser realizado para todos os pontos no vetor, tem-se um vetor ordenado.

O algoritmo *bucketSort* divide os pontos em 9 baldes, de acordo com o ângulo que formam com o ponto mais baixo (tal como nos outros métodos). Aqui, cada balde cobre uma faixa de ângulo de 20 graus entre 0° e 180° (primeiro balde: de 0° a 20°; segundo balde: de 20° a 40°). Vale ressaltar que, como os pontos são comparados com o ponto mais baixo do vetor, o maior ângulo possível é 180°. O método funciona da seguinte forma: primeiro, os pontos são colocados em seus respectivos baldes; depois, passa-se por cada um dos baldes e ordena-os utilizando o *insertionSort*; por fim, a função itera por todos os elementos de todos os baldes e os insere ordenadamente no vetor de pontos.

A função que implementa o Graham Scan constrói o fecho convexo utilizando os métodos de ordenação implementados previamente e uma pilha. Primeiro, é necessário encontrar o ponto mais baixo - isto é, aquele com a menor coordenada Y e, em caso de empate, com a menor coordenada X - e colocá-lo na primeira posição do vetor. Ele funcionará de pivô nesse algoritmo. Depois, seleciona-se um método a partir de um código passado como parâmetro para a função e ordena-se o vetor de pontos, tendo o primeiro ponto do vetor como pivô para as comparações. Após isso, é necessário remover pontos colineares, pois, para o fecho, eles são redundantes. Para isso, o vetor é percorrido e apenas os pontos mais distantes são mantidos.

Feito isso, cria-se uma pilha de pontos, passa-se por todos os pontos e, para cada um, verifica se o sentido da curva feita com os dois pontos anteriores é diferente de anti-horário - nesse caso, como o ponto ficará dentro do fecho e não fará parte do contorno, ele é removido. Esse processo continua até achar um ponto que faça uma curva no sentido anti-horário com os dois anteriores. Quando ele é encontrado, adiciona-o na pilha. Tal operação faz com que apenas os pontos mais externos do conjunto façam parte da pilha. Por fim, os pontos que formam o fecho são colocados no vetor responsável por armazená-lo.

Por fim, o método *jarvisMarch* implementa o Marchar de Jarvis e funciona da seguinte forma:

1. Encontra-se o ponto mais à esquerda e define-o como o ponto inicial.
2. Itera-se pelo conjunto de pontos.
3. Armazena-se o índice do ponto atual em um vetor de índices.

4. Para cada um dos pontos, vê-se qual o próximo ponto que faz a maior curva no sentido anti-horário em relação ao ponto atual e guarda esse ponto.
5. Define-se o ponto atual como o ponto encontrado.
6. Retorna-se ao passo 3 e repete esse processo até que o ponto atual seja igual ao ponto inicial - isto é, o ponto mais à esquerda.

Ao final desse processo, o vetor de índices contará com as posições no vetor de pontos de todos aqueles que formam o fecho convexo. Assim, basta passar esses pontos para o vetor que armazena o fecho.

Há também uma função auxiliar chamada *createHull*, que recebe um código como parâmetro e é usada apenas para determinar qual algoritmo será usado para formar o envoltório convexo.

É válido pontuar que todas as funções de ordenação utilizadas pelo scan de Graham utilizam como critério o ângulo que um ponto e o ponto pivô (que é aquele com a menor coordenada Y) formam com o eixo X.

3. Análise de complexidade

ccw(Point point1, Point point2, Point point3):

- Complexidade de tempo: O(1) - a função executa uma quantidade fixa de operações independentemente do tamanho dos pontos.

- Complexidade de espaço: O(1) - a função não utiliza estruturas de dados adicionais.

comparePoints(Point point0, Point point1, Point point2):

- Complexidade de tempo: O(1) - a função executa uma quantidade fixa de operações independentemente dos pontos.

- Complexidade de espaço: O(1) - a função não utiliza estruturas de dados adicionais.

mergeSort(Point *points, int begin, int end, Point pivot):

- Complexidade de tempo: O(n log n), onde n é o número de elementos no vetor a ser ordenado. O algoritmo de merge sort é usado de forma recursiva.

- Complexidade de espaço: O(n), onde n é o número de elementos no vetor a ser ordenado. É criado espaço adicional para os vetores auxiliares *valuesL* e *valuesR* usados na função **merge**.

insertionSort(int begin, int size):

- Complexidade de tempo: O(n^2), onde n é o número de elementos no vetor a ser ordenado. O algoritmo de Insertion Sort percorre o vetor várias vezes. Ele é indicado para vetores menores, mas tende a não ser tão rápido para um número grande de elementos.

- Complexidade de espaço: O(1) - a função não utiliza estruturas de dados adicionais.

bucketSort():

- Complexidade de tempo: O(n + k), onde n é o número de elementos no vetor a ser ordenado e k é o número de baldes. A fase de distribuição leva O(n) e a fase de junção dos baldes leva O(k).

- Complexidade de espaço: O(n + k), onde n é o número de elementos no vetor a ser ordenado e k é o número de baldes. São criados vetores auxiliares para armazenar os elementos em cada balde.

grahamScan(int sortingCode):

- Complexidade de tempo: O(n log n), onde n é o número de pontos a serem processados. A complexidade é dominada pela função de ordenação escolhida (*mergeSort*, *insertionSort* ou *bucketSort*).

- Complexidade de espaço: O(n), onde n é o número de pontos a serem processados. É alocado espaço para armazenar os pontos ordenados e o fecho convexo resultante.

jarvisMarch():

- Complexidade de tempo: O(nh), onde n é o número de pontos a serem processados e h é o número de pontos no fecho convexo.

- Complexidade de espaço: O(n), onde n é o número de pontos a serem processados. É alocado espaço para armazenar os índices dos pontos que farão parte do fecho convexo.

4. Estratégias de robustez

Para garantir a robustez do programa, as funções de ordenação e auxiliares usadas pelos algoritmos de construção do fecho foram implementadas como parte da classe *ConvexHull*. Essa decisão foi tomada tendo em vista que tais algoritmos são usados unicamente para ordenar conjuntos de pontos e auxiliarem na construção do envoltório convexo; logo, faz sentido que suas implementações sejam feitas específica e exclusivamente para isso.

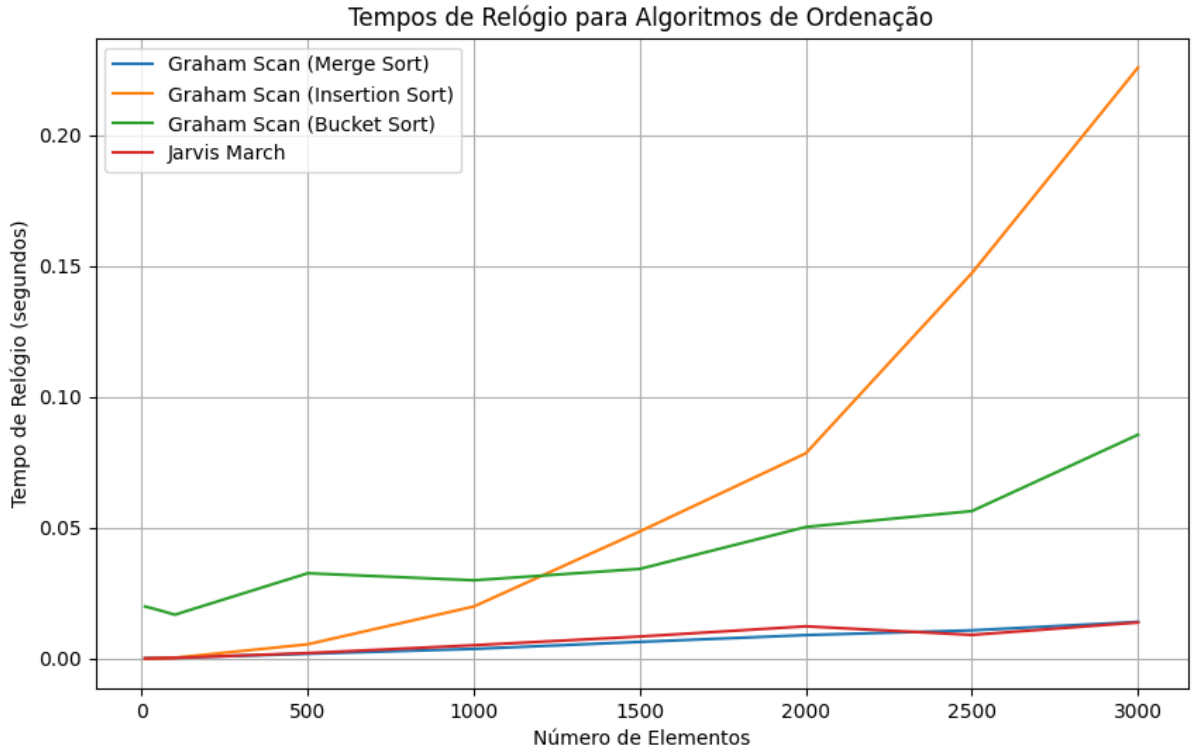
Além disso, presume-se que, se menos de três pontos são passados (ou obtidos após a remoção dos pontos colineares), não há como formar um polígono com eles. Assim, o número de pontos é checado antes da construção do fecho.

Para tratamento de eventuais erros, como o citado acima ou o caso de um arquivo que deveria ser aberto não for encontrado, foram criadas estruturas que representam exceções: *ExcecaoCodigoInvalido*, lançada caso o código de escolha da função de ordenação ou do algoritmo de construção do fecho, passado como parâmetro para a função *createHull*, não faça parte das opções possíveis; *ExcecaoPontosInsuficientes*, lançada caso o número de pontos no vetor não seja o suficiente para criar um polígono convexo; *ExcecaoArquivoNaoEncontrado*, caso um

arquivo inexistente seja passado como argumento na execução do programa; *ExcecaoDivisaoPorZero*, caso haja uma divisão por 0 no programa. Caso alguma dessas exceções seja lançada, a execução do programa é interrompida e ele é encerrado.

5. Análise experimental

Para mensurar o desempenho do programa, medi o tempo de relógio da execução de cada algoritmo para diferentes tamanhos de entrada, armazenei os resultados num arquivo .csv e, usando um programa em Python, criei o seguinte gráfico:



A análise do gráfico nos permite visualizar que as complexidades de tempo dos métodos bate com o exposto na seção “Análise de Complexidade”. Para casos onde o número de elementos é menor, o Insertion Sort se apresenta como o melhor dos métodos de ordenação; no entanto, quando o tamanho da amostra de dados começa a crescer, os outros métodos de ordenação performam melhor.

6. Conclusão

Tendo em vista o supracitado, o desenvolvimento de um algoritmo que constrói um fecho convexo utilizando os algoritmos Graham Scan e Jarvis March e métodos de ordenação já conhecidos foi realizado com sucesso.

Nesse trabalho, pude aprender bastante acerca de geometria computacional e como os algoritmos trabalham para construir fechos convexos. É interessante observar que há diferentes formas de se resolver esse problema complexo, e que cada uma pode ser usada de acordo com o contexto na qual se encontram e objetivo esperado. Além disso, minhas noções acerca da ordenação de diferentes amostras de dados foram muito ampliadas

7. Bibliografia

Slides da disciplina Estrutura de Dados disponibilizados pelo professor Wagner Meira. Disponíveis na metaturma de Estrutura de Dados na plataforma Moodle - UFMG. Acesso em 1 de junho de 2023.

Graham Scan. **Wikipedia**, 2023. Disponível em: https://en.wikipedia.org/wiki/Graham_scan. Acesso em 1 de junho de 2023.

Jarvis March. **Wikipedia**, 2023. Disponível em: https://en.wikipedia.org/wiki/Gift_wrapping_algorithm. Acesso em 1 de junho de 2023.

Geeks for Geeks. Disponível em: https://geeksforgeeks.org. Acesso em 3 de junho de 2023.

CPlusPlus. Disponível em: https://cplusplus.com/. Acesso em 2 de junho de 2023.

StackOverflow. Disponível em: https://stackoverflow.com. Acesso em 1 de junho de 2023.

8. Instruções para compilação e execução

Tendo já extraído o conteúdo do .zip em alguma pasta, faça o seguinte:

1 - Crie um arquivo chamado .txt e coloque nele os pontos a serem checados, um em cada linha e no formato “X Y” (coordenadas separadas por espaço).

2 - Abra uma janela do terminal na pasta que contém o trabalho.

3 - Execute o comando “make run”.

4 - Execute o comando “./bin/fecho <nome_do_arquivo.txt>”, substituindo <nome_do_arquivo.txt> pelo nome do arquivo que contém os pontos.

Obs.: Para remover os arquivos criados na execução do programa, execute o comando “make clean”.