UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Fecho Convexo

Aluno: Lucas de Oliveira Araújo

Matrícula: 333

Belo Horizonte, MG
2023

Sumário

---

1 Introdução

Um polígono é dito convexo se, e somente se, a reta determinada por dois vértices consecutivos quaisquer deixa todos os demais vértices num mesmo semiplano dos dois que ela determina [1]. Nesse sentido, um fecho convexo é um conceito geométrico que diz respeito ao menor polígono convexo formado a partir de um conjunto de pontos, de tal maneira que todos esses pontos estejam contidos no polígono [2]. O fecho convexo, por conseguinte, é o polígono de maior área e menor ângulo que contém todos os pontos do conjunto.

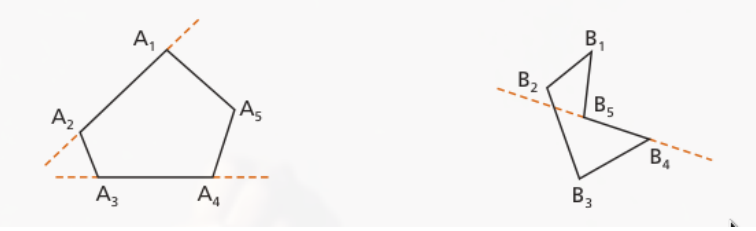

Figura 1: Os pontos A_n formam um polígono convexo, enquanto os pontos B_n formam um polígono côncavo (não convexo)

Na ciência da computação, o fecho convexo tem diversas aplicabilidades. Por exemplo, os veículos autônomos calculam o fecho convexo do veículo, isto é, o conjunto de pontos que o representa no espaço, para calcular as trajetórias seguras entre os obstáculos. Isso é feito pois, se nenhum ponto que forma o polígono convexo que representa o veículo colidir com o obstáculo, então nenhum outro ponto interno colidirá.

Nesse trabalho, vamos apresentar um programa que calcula o fecho convexo a partir de dois algoritmos computacionais, a saber, o Graham Scan e o Jarvis March.

2 Método

2.1 Tipos abstratos de dados

Dado a natureza geométrica do problema, fica evidente a necessidade de representar os objetos geométricos básicos. Nesse sentido, foram implementadas os tipos abstratos de dados `Point2D`, responsável por representar um ponto no plano, e `Line2D`, responsável pela representação de um segmento de reta.

Além disso, os métodos de geometria analítica, isto é, cálculo de distância entre dois pontos, ângulo polar etc, foram implementados em uma classe estática denominada `Utils`. O TAD `ConvexHull`, também uma classe estática, contém os métodos que executam os algoritmos Graham Scan e Jarvis March.

Por fim, por se tratar de um programa fundamentalmente geométrico, é de grande valia conseguir visualizar de forma gráfica a representação das figuras durante a execução dos algoritmos citados. Tendo isso em vista, uma classe denominada `AnimationController` foi criada para manipular toda a parte gráfica do programa, a qual foi possível com o uso da biblioteca SFML[1].

[1] `https://www.sfml-dev.org/`

2.2 Estruturas de dados

Sendo um polígono convexo um conjunto de pontos, se faz necessário um meio para agrupar tais pontos. Para tanto, implementamos a estrutura de dados `Vector`. O intuito dessa implementação, além de representar o conjunto de pontos, é facilitar a manipulação destes.

A versatilidade do Vector permitiu ser esta a única estrutura de dados necessária para a execução do programa, além do array básico da linguagem C++.

3 Análise de Complexidade

3.1 Complexidade de tempo

Quando o programa recebe um conjunto de pontos, a sua primeira ação é armazená-los em um vector, com um custo resultante de $O(n)$, onde n é a quantidade de pontos.

Como próximo passo, é executado os algoritmos Graham Scan e Jarvis March. Com relação ao Jarvis, é notório que a sua complexidade de tempo é $O(nh)$, onde n é a quantidade de pontos no conjunto e h é o número de vértices necessários para formar o fecho convexo.

Já com relação ao Graham Scan, a sua peculiaridade é que o conjunto de pontos deve estar ordenado consoante o ângulo polar em relação a algum ponto de referência para que o algoritmo encontre o fecho convexo. Consequentemente, é necessário utilizar algum algoritmo de ordenação para satisfazer esse requisito.

Dentro dessa perspectiva, os algoritmos escolhidos foram:

1. Merge Sort, com complexidade de tempo $O(n \log n)$, no pior, médio e melhor caso
2. Insertion Sort, com complexidade de tempo $O(n^2)$ no pior e médio caso, e $O(n)$ no melhor
3. Bucket Sort, com complexidade de tempo $O(n)$ no caso médio, o que ocorre quando o número de "baldes"tende a n, e $O(n^2)$ no pior caso[3]

Dado o conjunto de pontos já ordenado, o Graham Scan tem complexidade de tempo $O(n)$ [4, 2]. Entretanto, em geral o algoritmo precisará ordenar os pontos, o que faz com que a sua complexidade dependa da complexidade do algoritmo utilizado na ordenação. Portanto, considerando os casos médios dos algoritmos de ordenação, temos que:

- Graham + Merge: $O(n \log n)$
- Graham + Insertion: $O(n^2)$
- Graham + Bucket: $O(n)$

3.2 Complexidade de espaço

No tocante ao espaço de memória utilizado durante a execução do programa, citemos a estrutura de dados Vector. Nessa estrutura, a complexidade de espaço é $O(n)$, onde n é proporcional ao número de elementos armazenados. Dizemos proporcional, pois o vector implementado precisa alocar certo espaço a mais para possibilitar algum crescimento futuro e, quando o espaço alocado é totalmente preenchido, um vector maior é criado e os elementos são copiados para ele.

Dentro dessa lógica, os algoritmos implementados devem receber dois vectors por referência, sendo um com os pontos que devem ser analisados pelo algoritmo e o outro um vector vazio no qual será inserido sequencialmente o conjunto de pontos que forma o polígono convexo. Por conseguinte, o pior caso seria àquele em que todos os pontos do conjunto são utilizados como vértices do fecho convexo, conforme ilustrado na figura 2. Nesse caso em específico, a complexidade de espaço seria $O(n)+O(n) = O(n)$, onde n é a quantidade de pontos recebidos pelo algoritmo. O melhor caso seria aquele em que um conjunto de n pontos tem o fecho convexo formado por apenas 3 pontos (menor número de pontos não colineares necessários para formar um polígono). Nesse caso a complexidade de espaço também seria $O(n)$, uma vez que $O(n) + O(3) = O(n)$.

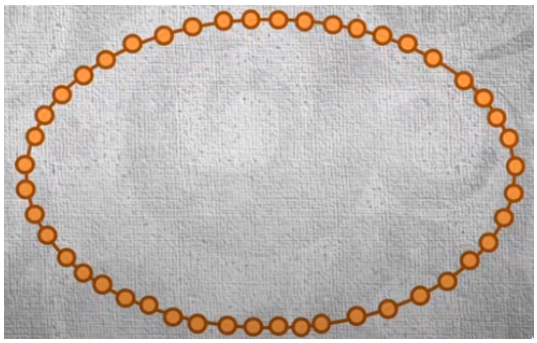

Figura 2: Pior caso de complexidade de espaço

Portanto, a complexidade de espaço do Graham Scan e do Jarvis March implementados para este programa é $O(n)$.

4 Estratégias de Robustez

A estratégia de robustez empregada foi o controle de exceções, além dos testes implementados utilizando a biblioteca DOCTEST[2].

O controle de exceções foi implementado em múltiplas partes do programa. Para cada uma das estruturas de dados utilizadas, existe um conjunto de exceções que são lançadas e tratadas, conforme o necessário. Dessa maneira, erros malquistos são devidamente tratados, de forma que a experiência do usuário não seja desagradavelmente afetada.

Os testes foram implementados considerando o mais diversos tipos de cenários. Da mesma maneira que as exceções, para cada estrutura de dados implementada, há também um conjunto de testes realizados no intuito de confirmar a resiliência de tais estruturas.

Ademais, para as principais classes do programa, a saber, Point2D, Line2D, Utils e ConvexHull, também foram implementados um conjunto de testes multifocais, visto que seus métodos estão no cerne de todo o funcionamento do código. Dada essa importância, tais classes não poderiam ser utilizadas sem antes a confirmação de que elas trabalhariam conforme o esperado.

[2]DOCTEST pode ser encontrado no github: `https://github.com/doctest/doctest`

5 Análise Experimental

5.1 Analise do tempo de execução

Uma vez que todo o programa gira em torno dos algoritmos de busca do fecho convexo, decidimos direcionar nossos experimentos para tais algoritmos. Para facilitar a execução dos testes, uma classe estática denominada `Analyzer` foi implementada, a qual controla os testes desde a sua execução até a plotagem dos gráficos.

O experimento consistiu em verificar o tempo decorrido na execução dos dois algoritmos, considerando também as três versões do Graham Scan (com Merge Sort, Insertion Sort ou Bucket Sort). Os testes foram feitos para conjuntos entre 100 e 2000 pontos, variando em 100 o número de pontos. Além disso, tais testes foram conduzidos em baterias de 10 testes para cada conjunto de n pontos, com o intuito de diluir possíveis outliers, isto é, pertubações nas amostras coletadas. Os testes são apresentados na figura 3.

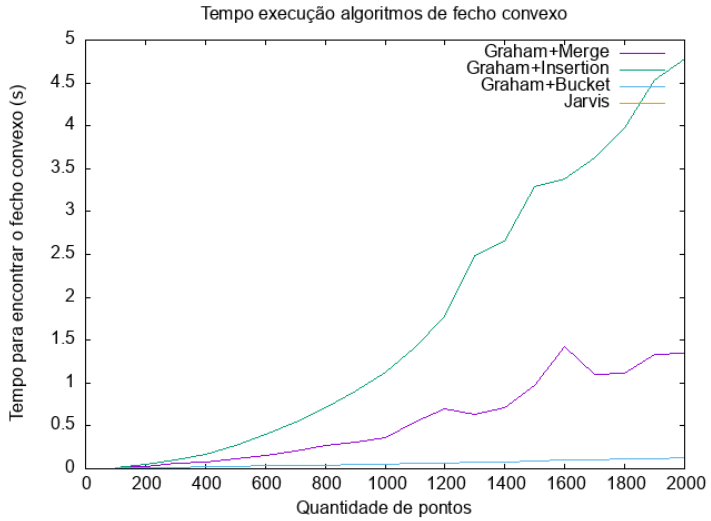

Figura 3: Testes de tempo de execução

Em primeiro lugar, o algoritmo que obteve maior performance foi o Jarvis March, encontrando o fecho convexo em um conjunto 2000 pontos em apenas 0.0054s. Dessa maneira, a sua linha no gráfico da figura 3 foi desenhada basicamente em cima do eixo x, o que explica a sua aparente ausência.

Um segundo ponto interessante é o crescimento acentuado do tempo de execução do Graham com Insertion Sort. Esse crescimento condiz com a análise de complexidade de tempo realizada anteriormente, na qual foi concluído que o uso do Insertion Sort no algoritmo do Graham Scan ocasionava uma complexidade de $O(n^2)$. Além disso, as demais curvas do Graham Scan com Merge Sort e do Graham Scan com Bucket também estão de acordo com a referida análise de complexidade.

Portanto, através da análise de complexidade de tempo realizada e dos testes de tempo de execução, fica evidente que a melhor combinação é a Graham+Bucket. Além do mais, o algoritmo do Jarvis March apresenta performance superior a qualquer combinação do Graham Scan ao longo de todo o intervalo de quantidade de pontos considerado.

6 Conclusões

Nesse trabalho foi proposto um programa que recebe um conjunto de pontos no plano e encontra o seu fecho convexo. Tal programa utiliza-se das estruturas de dados Vector e array básico do C++ para manipular e armazenar os pontos, além de dispor-se de um ferramental completo no tocante a tratativa de possíveis erros.

Por fim, a construção do referido programa possibilitou um amplo aprendizado sobre a manipulação das estruturas de dados citadas, além de reforçar os conceitos básicos de programação orientada a objetos. Além disso, a implementação do referido programa exigiu o estudo dos algoritmos Graham Scan e Jarvis March, além da biblioteca SFML, responsável pela parte gráfica do programa. Não menos importante, para a execução acurada do Graham Scan, o conjunto de pontos deve estar ordenado. Consequentemente, foi necessário o estudo dos vários algoritmos de ordenação disponíveis.

Referências

[1] Gelson Iezzi. *Fundamentos De Matematica Elementar - Geometria plana - Vol.8*. Atual, 2016.

[2] Thomas H. Cormen et al. *Algoritmos - Teoria e Prática*. Elsevier Brasil, 2017. ISBN: 978-85-3527-179-9.

[3] *Bucket Sort*. URL: `https://www.javatpoint.com/bucket-sort` (acesso em 08/06/2023).

[4] *Convex Hull Algorithms - Graham Scan*. URL: `https://algorithmtutor.com/Computational-Geometry/Convex-Hull-Algorithms-Graham-Scan/` (acesso em 05/06/2023).

[5] *SFML API Documentation*. URL: `https://www.sfml-dev.org/learn.php` (acesso em 10/06/2023).

Apêndice

Instruções para compilação e execução

Uma vez no diretório raiz do programa, a compilação poderá ser realizada por meio do comando `make build`.

OBS.: O programa utiliza-se da biblioteca SFML para gerar as representações gráficas. Isso implica que esta biblioteca deve estar disponível em seu computador para que o processo de compilação ocorra sem erros. Para instalar a referida biblioteca em sistemas derivados de Debian, execute `sudo apt-get install libsfml-dev`. Para sistemas derivados de Arch, execute `sudo pacman -S sfml`. Demais instruções estão disponíveis em [5].

Após a compilação, o programa poderá ser executado com o comando `make run`. Nessa etapa, você terá algumas possibilidades:

1. Execução com passagem de parâmetro: `make run ARGS="<params>"`. Os parâmetros disponíveis estão expostos na figura 4.
2. Execução sem passagem de parâmetro: `make run`
   (a) O programa será iniciado e aguardará o caminho do arquivo com o conjunto de pontos
      i. Esse caminho pode ser passado digitando `fecho caminho/entradas.txt` ou somente `caminho/entradas.txt`
   (b) A execução sem parâmetro pode ser agilizada quando houver disponível um arquivo de texto de forma que cada linha desse arquivo contenha o caminho de um arquivo com as entradas. Dessa forma, você pode simplesmente utilizar o operador de direcionamento dos sistemas Unix da seguinte maneira: `make run < caminho/todas_as_entradas.txt`

```
FECHO CONVEXO

  -f, --file <string>   Caminho do arquivo
  -r, --random          Gera pontos aleatórios
  -p, --points <int>    Número de pontos aleatórios
  -g, --graphic         Habilita o modo gráfico
  -a, --analyzer        Realiza uma bateria de testes
  -h, --help            Mensagem de ajuda
```

Figura 4: Parâmetros disponíveis

Ademais, os testes e o valgrind podem ser executados via os comandos `make tests` e `make valgrind`, respectivamente.

Instruções para uso do modo gráfico

O modo gráfico é habilitado por meio da flag `-g`. Após a execução e analise dos tempos de execução dos algoritmos, uma interface gráfica será aberta. Algumas instruções para a execução das animações seguem abaixo:

1. Arrastar a tela:
   (a) Clique com o botão direito do mouse para ativar o modo de arrasto (não precisa manter pressionado)

(b) Clique novamente com o botão direito para desativar esse modo

2. Zoom

   (a) Role a roda do mouse para aumentar ou diminuir o zoom

3. Animações

   (a) Pressione a tecla 'g' para executar a animação do Graham Scan

   (b) Pressione a tecla 'j' para executar a animação do Jarvis March

   (c) Ao fim de uma animação, você pode pressionar alguma das teclas acima para iniciar uma nova execução

4. Finalizar

   (a) Pressione a tecla 'q' para fechar a tela de animação

5. Dica

   (a) Enquanto a animação do Graham Scan ou do Jarvis March estiver rodando o zoom e o arrasto de tela não funcionam. Portanto, sugiro, se necessário, posicionar a tela da melhor maneira antes de executar as animações

   (b) A captura dos eventos gerados pelo acionamento das teclas 'g', 'j' ou 'q' é cumulativo, isto é, durante uma animação o pressionamento destas teclas agendará a respectiva ação para quando a animação vigente finalizar