Trabalho Prático

Fecho Convexo

Thales Augusto Rocha Fernandes - thalesarf@ufmg.br

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais

1. Introdução

Nesse trabalho é tratado o problema do fecho convexo que é o menor polígono convexo que encapsule todos os pontos escolhidos em um dado plano. Existem dois algoritmos famosos para solucionar esse problema que são o Scan de Graham e a Marchar de Jarvis, sendo que o Scan de Graham requer um método de ordenação para funcionar corretamente. Dessa maneira, o objetivo desse trabalho é criar um código em linguagem C/C++ que recebe como entrada uma série de coordenadas de pontos cartesianos e, a partir desses pontos, realiza o fecho convexo uma vez com a Marchar de Jarvis e três vezes com o Scan de Graham, de forma que são usados diferentes algoritmos de ordenação para cada chamada do Scan de Graham. Os algoritmos de ordenação usados são o MergeSort, o InsertionSort e o RadixSort/BucketSort, de maneira que a execução do Marchar de Jarvis e de cada Scan de Graham será cronometrado. Assim, será possível saber quais algoritmos são mais eficientes do que os outros, tanto na questão do fecho convexo que são as duas soluções centrais, quanto na questão dos métodos de ordenação. Portanto, ao final do código será apresentado o fecho convexo resultante e o tempo gasto por cada uma das quatro soluções do problema. Essa implementação não fará uso de estruturas de dados provenientes de bibliotecas específicas da linguagem C++. Assim, a representação da estrutura de dados usadas estão disponíveis no código e possuem funções e características específicas para este projeto.

2. Método

2.1. Configurações da máquina

Sistema operacional: WSL2 - Ubuntu 20.04 LTS

2.2. Estruturas de dados

Na implementação desse trabalho, as estruturas de dados usadas foram a pilha, o Ponto, a Reta e o Fecho que são classes que serão explicadas no tópico 2.3.

2.3. Classes

A classe Ponto possui duas variáveis double privadas chamadas x e y, que representarão as coordenadas no plano cartesiano de um ponto qualquer. As operações públicas que essa classe realiza são criar Ponto, retornar a coordenada x, retornar a coordenada y, atribuir um valor à coordenada x, atribuir um valor à coordenada y e um destrutor. A classe Reta possui duas estruturas Ponto como atributos privados que representam os dois pontos extremos que formam a reta. As operações públicas que essa classe realiza são criar Reta, retornar o ângulo formado pela Reta, retornar o comprimento da Reta e um destrutor. A classe Fecho representa um array dinâmico,

ou seja, é capaz de crescer ou diminuir automaticamente conforme necessário e também foi criado com um template. Essa classe faz parte do código devido à facilidade de armazenar e manipular a estrutura de dados Ponto que iremos criar, de forma que o Fecho será usado para armazenar todos os pontos que formam o fecho convexo na saída das funções que serão especificadas no tópico 2.4. As operações que esse Fecho realiza são criar um novo Fecho, Destruir o Fecho, retornar o tamanho atual do Fecho, retornar se o Fecho estiver vazio, adicionar um elemento ao final do Fecho, acessar um elemento no índice especificado, retornar o elemento do índice especificado e um operador de atribuição. A classe PilhaArranjo faz parte do código em vista da sua característica de que o último elemento a ser inserido nessa estrutura é o primeiro a ser retirado, o que é essencial para a implementação do algoritmo Scan de Graham. Essa pilha é do tipo sequencial com o uso de arranjos e alocação estática e a pilha usa um template para que ela possa armazenar itens em qualquer formato, inclusive na forma da estrutura de dados Ponto. Dessa maneira, a classe PilhaArranjo será usada para armazenar o fecho convexo resultante das funções que realizam o Scan de Graham, o que vai ser explicado com mais detalhes no tópico 2.4. As operações que essa pilha realiza são criar uma nova pilha, testar se a pilha está vazia, empilhar um item, desempilhar um item, limpar a pilha e retornar o item do topo da pilha.

2.4. Funções

O código possui 5 principais funções que estão separadas em diferentes arquivos que possuem o mesmo nome de cada função principal, com exceção da terminação “.cpp” e “.h” contida nesses arquivos.

main: Essa função lê o arquivo de entrada do usuário e ativa as outras funções do programa para realizar o código proposto na introdução. Para isso, foi criado 3 strings chamadas entrada, x e y, de forma que, enquanto for possível ler o arquivo de entrada ele será armazenado na string entrada. Cada linha do arquivo de entrada fornece as duas coordenadas cartesianas de um ponto e, assim, após ler uma linha com a string entrada o primeiro valor inteiro é atribuído à string x e o segundo valor à string y. Depois, essas strings x e y são convertidas para o tipo double através da função stod e são armazenadas em uma estrutura Ponto, de maneira que todos esses Pontos formados na entrada serão armazenados em um Fecho chamado Jarvis_Entrada. A classe Fecho só é usada para armazenar a entrada para o algoritmo Marcha de Jarvis, de forma que os algoritmos scan de Graham terão suas entradas armazenadas em um array da classe Ponto. A classe Fecho será usada prioritariamente para armazenar o fecho convexo resultante das funções que explicaremos a seguir. Após ler o arquivo de entrada o código ativa as outras 4 funções principais que realizam os fechos convexos e mede o tempo que cada uma delas gasta. Essa cronometragem do tempo gasto é realizada através da biblioteca chrono. Por fim, é imprimido no arquivo de saída os pontos que formam o fecho convexo e os tempos medidos anteriormente.

MarchaJarvis: Essa função realiza o algoritmo de Marchar de Jarvis no qual é recebido um Fecho chamado Jarvis_entrada contendo os pontos como parâmetro. Para isso, a função encontra o ponto mais à esquerda e mais embaixo entre todos os pontos que é o ponto inicial e salva o índice desse ponto, já que esse ponto sempre

estará em qualquer fecho convexo. A partir desse ponto, a função percorre todos os pontos restantes e encontra o próximo ponto que contém uma orientação anti-horária em relação ao ponto atual, sendo que há uma função chamada orientação que faz esse cálculo. Esse ponto encontrado se torna o ponto atual e o mesmo processo é repetido até que seja encontrado o ponto inicial, de maneira teremos encontrado todos os pontos do fecho convexo. Esses pontos encontrados são armazenados dentro de um Fecho chamado Jarvis_saida que é passado por referência na função.

Graham_Merge: Essa função realiza o algoritmo de Scan de Graham com o auxílio do método de ordenação mergeSort, de maneira que é recebido um array da classe Ponto chamado Merge_entrada contendo os pontos como parâmetro, uma variável int que indica o tamanho desse array e um Fecho chamado Resposta que irá guardar os pontos que formam o fecho como parâmetro de referência. Para isso, a função inicialmente encontra o ponto mais à esquerda e mais embaixo entre todos os pontos que é o ponto inicial e salva o índice desse ponto, já que esse ponto sempre estará em qualquer fecho convexo. Após achar esse ponto inicial, o algoritmo faz a ordenação dos pontos a partir do mergeSort e, posteriormente, dentre esses pontos ordenados o algoritmo verifica se há pontos colineares, dos quais alguns terão que ser removidos. Finalmente, iremos usar uma PilhaArranjo de tipo Ponto, de forma que os 2 primeiros itens já são incluídos na pilha e para cada ponto subsequente na lista ordenada, é verificado se ele faz uma curva à esquerda ou uma curva à direita em relação aos dois pontos no topo da pilha. Isso é feito usando a função de orientação para determinar a orientação dos pontos. Se o ponto faz uma curva à esquerda, ele é adicionado à pilha. Caso contrário, se o ponto faz uma curva à direita, isso indica que o ponto anterior na pilha não faz parte da envoltória convexa. Nesse caso, o ponto anterior é removido da pilha e o processo é repetido até que o ponto atual faça uma curva à esquerda com os pontos no topo da pilha. Para verificar essas curvas e ângulos nós usamos a classe Reta na função auxiliar Compara_Merge que compara o ângulo entre pontos. Esse processo é repetido para todos os pontos subsequentes na lista ordenada até que a pilha contenha o fecho convexo, de forma que o ponto inicial está no começo da pilha é o último ponto do fecho percorrido em rotação anti-horária. Finalmente, todos os itens da pilha são jogados dentro do Fecho chamado Resposta que é passado por referência.

Graham_Insertion: Essa função realiza o algoritmo de Scan de Graham com o auxílio do método de ordenação insertionSort, de maneira que é recebido um array de Ponto chamado pontos contendo os pontos de entrada, um int que indica o tamanho desse array e um Fecho chamado Resposta que irá guardar os pontos que formam o fecho como parâmetro de referência. Todas as operações feitas na função Graham_Merge são idênticas às feitas nessa função, exceto o algoritmo de ordenação.

Graham_Linear: Essa função realiza o algoritmo de Scan de Graham com o auxílio do método de ordenação radixSort se a entrada tiver mais que 10 pontos e com o auxílio do bucketSort com mergeSort se a entrada tiver menos que 10 pontos. Isso ocorre, pois o radixSort possui 10 vetores para serem ordenados e não é ideal quando há menos de 10 elementos para serem ordenados. Assim, essa função recebe um array de Ponto chamado pontos contendo os pontos de entrada, um int que indica o tamanho desse array e um Fecho chamado Resposta que irá guardar os pontos que formam o

fecho como parâmetro de referência. Todas as operações feitas na função Graham_Merge são idênticas às feitas nessa função, exceto o algoritmo de ordenação.

3.Análise de Complexidade

Nessa análise, vamos considerar o valor n como o número de pontos contidos na entrada.

Main – Complexidade de tempo: Essa função possui 2 loops que percorrem todos os pontos do arquivo de entrada. Dessa forma, sua complexidade é O(n).

Main – Complexidade de Espaço: Essa função cria quatro estruturas que armazenam todos os pontos de entrada. Dessa forma, sua complexidade é O(n).

MarchaJarvis – Complexidade de tempo: Essa função possui um loop externo que percorre todos os pontos de entrada para encontrar o ponto mais à esquerda. Esse loop tem uma complexidade de tempo de O(n). Em seguida, o algoritmo tem um loop interno que percorre os pontos da envoltória convexa. Esse loop é executado h vezes, onde h é o número de pontos na envoltória convexa. Dentro desse loop, há um loop adicional que percorre todos os pontos de entrada para encontrar o próximo ponto na envoltória convexa. Esse loop tem uma complexidade de tempo de O(n). Portanto, a complexidade assintótica de tempo dessa função é O(nh), onde n é o número total de pontos de entrada e h é o número de pontos na envoltória convexa.

MarchaJarvis – Complexidade de espaço: essa função requer espaço adicional para armazenar os pontos de entrada, a envoltória convexa resultante e algumas variáveis auxiliares. Assumindo que o espaço necessário para armazenar os pontos de entrada é proporcional a n e o espaço para armazenar a envoltória convexa é proporcional a h, a complexidade assintótica de espaço é O(n), já que o maior caso de h é quando o fecho resultante possui todos os pontos de entrada que é igual a n.

Graham_Merge – Complexidade de tempo: A complexidade assintótica de tempo do código fornecido é dominada pelo algoritmo de ordenação mergeSort, que é chamado para ordenar os pontos antes de encontrar a envoltória convexa. O restante das operações no código, como encontrar o ponto com menor coordenada y e empilhar os pontos na envoltória convexa, são operações de tempo linear em relação ao número de pontos. Como temos 2 loops dentro da função, considerando o tempo do mergeSort, a função possui uma complexidade de (3n log n). Dessa forma, a complexidade assintótica de tempo é O(n log n) que é a mesma complexidade do mergeSort, mesmo com a função de comparação.

Graham_Merge – Complexidade de espaço: Essa função recebe e utiliza as classes Ponto[], Fecho e PilhaArranjo para armazenar e manipular os pontos. Para realizar a ordenação é criado mais um vetor de Ponto auxiliar. Como o Ponto[] e o Fecho irão armazenar no máximo n pontos, a complexidade de espaço de tempo será uma constante multiplicado por n, ou seja, O(n).

Graham_Insertion – Complexidade de tempo: O código possui loops para encontrar o ponto com menor coordenada y colineares e para empilhar os pontos na envoltória convexa, de forma que essas operações tem complexidade de tempo linear igual a O(n). A complexidade de tempo do insertionSort é O(n^2) sem considerar a função de comparação. Essa função de comparação por calcular ângulos e distâncias pode ser

custosa e, dessa forma, a complexidade de toda a função será O(n^2 log n) já que essa função de comparar realmente impacta a eficiência do insertionSort.

Graham_Insertion – Complexidade de espaço: Essa função recebe e utiliza as classes Ponto[], PilhaArranjo e Fecho para armazenar e manipular os pontos. Como essas irão armazenar no máximo n pontos, a complexidade de espaço de espaço será uma constante multiplicado por n, ou seja, O(n).

Graham_Linear – Complexidade de tempo: Essa função realiza um loop percorrendo todos os pontos de entrada para achar o ponto inicial, também realiza um loop entre os pontos ordenados para eliminar pontos colineares e empilhar os pontos na pilha. As complexidades anteriores são O(n). No entanto, também temos a complexidade do método de ordenação radixSort. Esse algoritmo é variável dependendo da implementação e, nesse caso, possui um número de dígitos de pontos igual a 6, se considerarmos o número 100000 e possui 10 baldes. Segundo a definição desse método de ordenação, a complexidade seria O(5 * (n + 10)), de acordo com os valores comentados acima. No entanto, também é realizado muitos loops dentre as retas e ângulos comparados no algoritmo. Dessa maneira, a complexidade dessa função é O(n log n).

Graham_Linear – Complexidade de espaço: Essa função recebe e utiliza as classes Ponto[], PilhaArranjo e Fecho para armazenar e manipular os pontos. Como essas irão armazenar no máximo n pontos, a complexidade de espaço de espaço será uma constante multiplicado por n, ou seja, O(n).

4. Análise de Robustez

Os erros tratados nesse projeto foram o erro de abertura de arquivo, erro de pontos insuficientes e erro de pontos colineares. Se o arquivo especificado para leitura não existir, o código avisa o usuário e encerra o programa. Caso a leitura seja possível, o código verifica se conjunto de pontos da entrada tem menos que 3 pontos e, se for o caso, o código avisa o usuário e encerra o programa, já que não é possível realizar um fecho convexo com menos de 3 pontos. Se o conjunto de pontos da entrada tiver menos que 3 pontos não colineares, o código avisa o usuário e encerra o programa, já que não é possível realizar um fecho convexo com menos que 3 pontos não colineares.

5. Análise Experimental

Iremos analisar o tempo gasto pelos 4 algoritmos que realizam o fecho convexo, por meio das funções de medição de tempo da biblioteca chrono e por meio da ferramenta de gráficos gnuplot. Para o teste das funções MarchaJarvis, Graham_Merge e Graham_Linear nós realizamos 10 testes variando o número de pontos entre 1000000 e 10000000. Já para o teste do Graham_Insertion que demanda bem mais tempo que as outras funções nós realizamos 10 testes variando entre 10000 a 28000 pontos. Os testes revelaram que a eficiência das funções MarchaJarvis, Graham_Merge e Graham_Linear é bem parecida e qualquer uma das três é ótima para a solução do problema, de maneira que o Marcha_Jarvis é o melhor, por possuir a menor complexidade de tempo. O resultado do Graham_Insertion é péssimo, já que o número de pontos teve que ser diminuído drasticamente para que o parâmetro de tempo gasto fosse um pouco parecido com o das outras funções, de forma que essa função não seja recomendável para solucionar o problema. A seguir, estão os gráficos que demonstram esses experimentos:

Gráfico do MarchaJarvis:

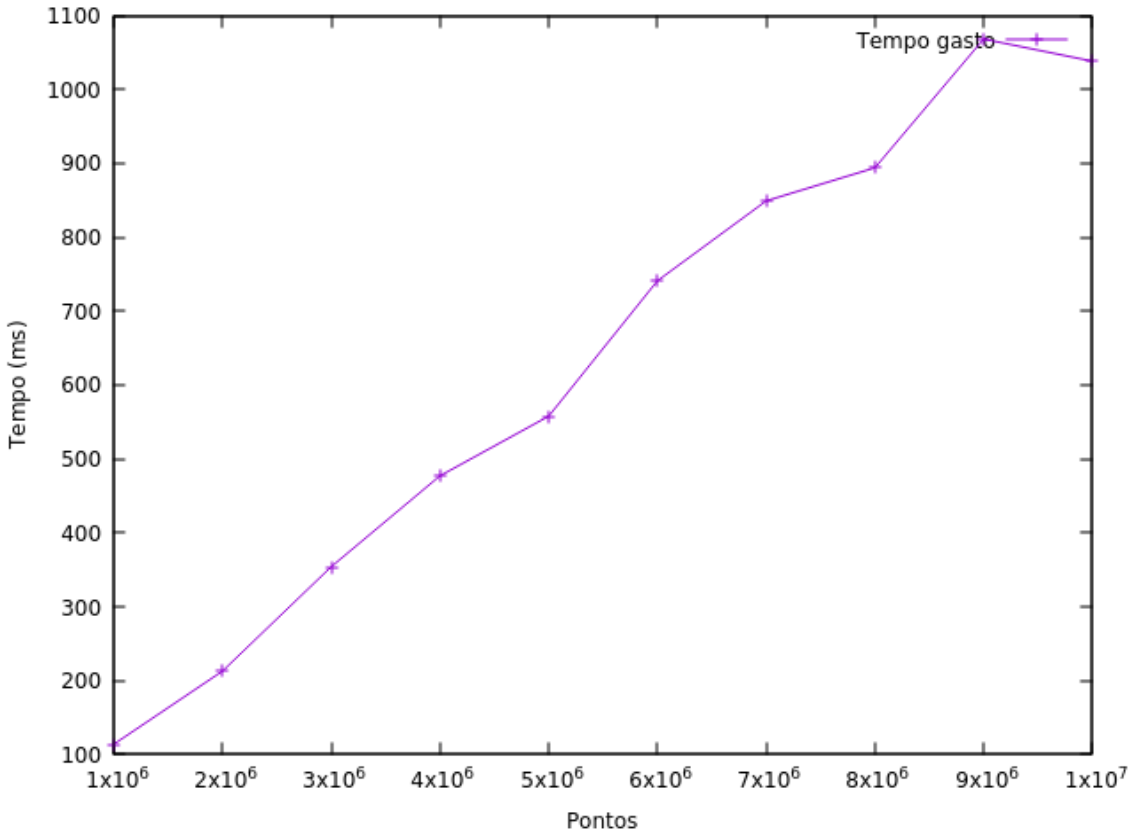


Gráfico Graham_Merge:

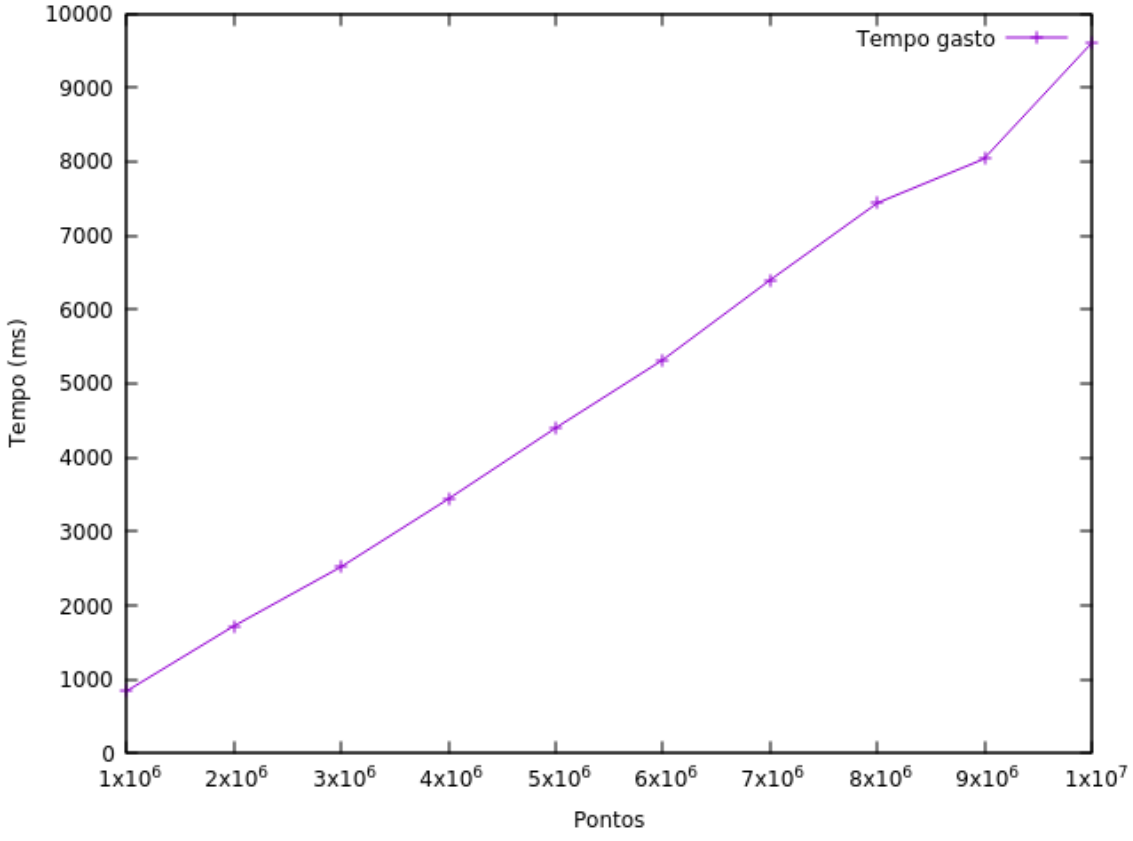


Gráfico Graham_Linear:

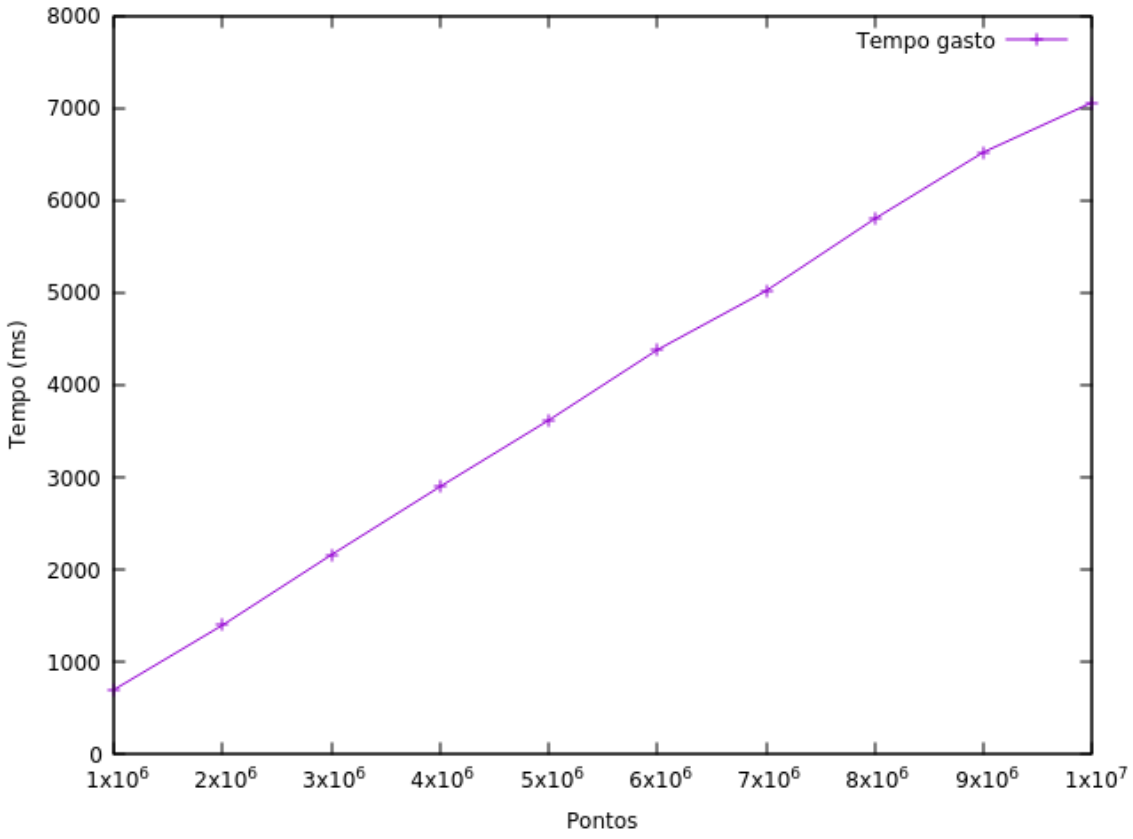
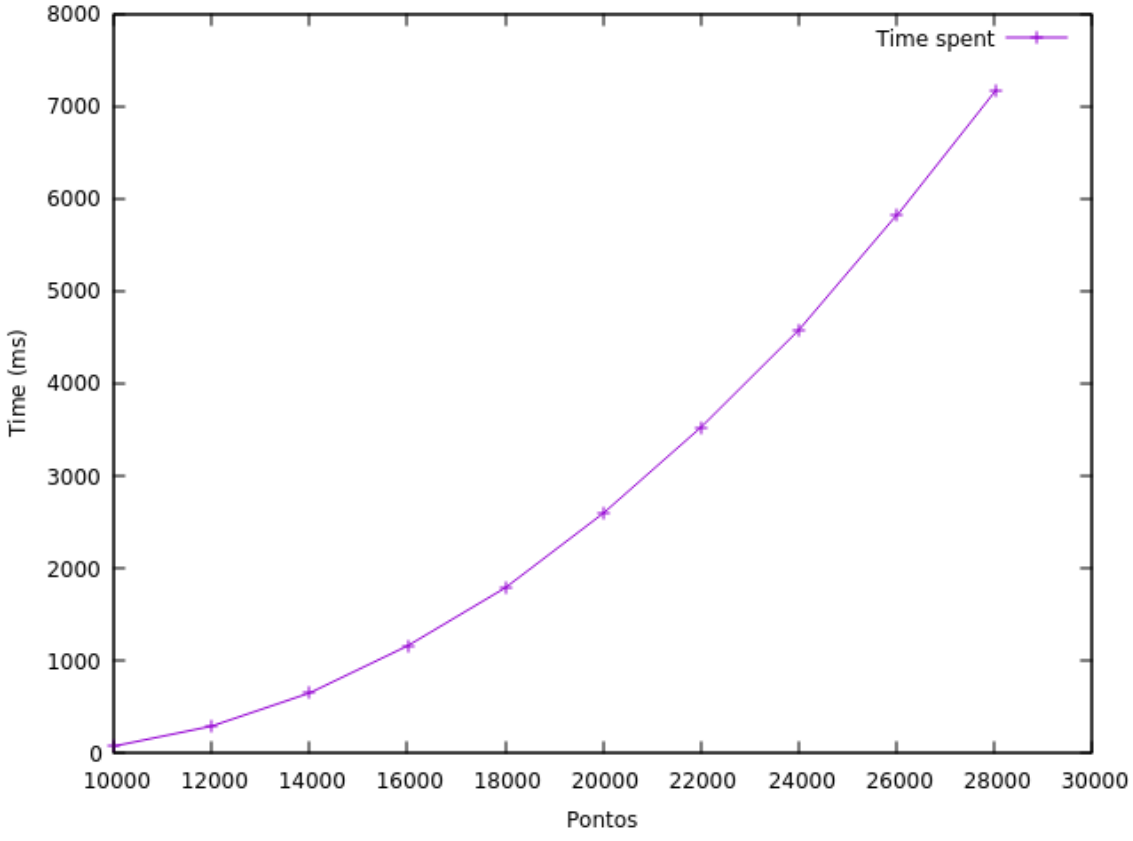


Gráfico Graham_Insertion:



Conclusão:

Nesse trabalho foi realizado um código na linguagem C++ que calcula o fecho convexo de um número de pontos através dos algoritmos Marchar de Jarvis e Scan de Graham no qual também foi utilizado os métodos de ordenação mergeSort, insertionSort e Radixsort. Esse programa foi realizado sem utilizar estruturas de dados provenientes de bibliotecas externas do C++, de forma que as estruturas Ponto, Reta, Fecho e PilhaArranjo foram implementadas e possuem funções e características específicas para este projeto. Dessa maneira, o processo de formulação do código proporcionou um maior entendimento acerca dos tipos abstratos de dados e um maior entendimento sobre os algoritmos de ordenação utilizados. Além disso, esse projeto demandou o uso de uma quantidade considerável de funções com um certo nível de dificuldade, o que exigiu o uso de ferramentas de depuração como o GDB e o valgrind. Outro componente exigido por esse trabalho foi a análise de complexidade, por meio de notações assintóticas, e a análise experimental, por meio de ferramentas que testam os desempenhos computacionais. A utilização de conceitos e ferramentas usados para compreender as estruturas de dados, as estratégias de depuração e as estratégias de análise experimental e de análise de complexidade foram ensinados nas aulas de Estrutura de Dados e, assim, esse trabalho prático foi essencial para fixar o conteúdo do curso e para aumentar a familiaridade com projetos de programação.

Bibliografia

Livro Thomas H. Cormen - Algoritmos: Teoria e Prática

Slides virtuais da disciplina de Estrutura de Dados disponibilizados por Wagner Meira Jr.

Instruções para Compilação e Execução:

A compilação e execução é feita através do makefile.

O usuário deve deixar o arquivo txt que será lido pela entrada dentro da pasta TP que é a raiz do projeto. Ao inserir esse arquivo, ele deve digitar no terminal o comando “make fecho=arquivo.txt”, sendo que o “arquivo” do comando deve ser substituído pelo nome do arquivo de entrada.

Se o usuário desejar executar o código manualmente pelo terminal, após utilizar o comando make, ele deve digitar “./bin/TP arquivo.txt” e novamente substituir “arquivo” pelo nome do arquivo de entrada.

Os dois métodos anteriores imprimem a saída na saída padrão, como descrito nas especificações do trabalho.