Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Sweep Line: Um Paradigma para Solução de Problemas de Geometria Computacional

Autora: Sara Conceição de Sousa Araújo Silva
Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF
2021

Sara Conceição de Sousa Araújo Silva

Sweep Line: Um Paradigma para Solução de Problemas de Geometria Computacional

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2021

Sara Conceição de Sousa Araújo Silva

Sweep Line: Um Paradigma para Solução de Problemas de Geometria Computacional/ Sara Conceição de Sousa Araújo Silva. – Brasília, DF, 2021-

97 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2021.

1. *sweep line*. 2. geometria computacional. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. *Sweep Line*: Um Paradigma para Solução de Problemas de Geometria Computacional

CDU 02:141:005.6

Sara Conceição de Sousa Araújo Silva

Sweep Line: Um Paradigma para Solução de Problemas de Geometria Computacional

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 03 de novembro de 2021:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. John Lenon Cardoso Gardenghi
Convidado 1

Prof. Dr. Vinicius Ruela Pereira Borges
Convidado 2

Brasília, DF
2021

Agradecimentos

Agradeço a minha mãe e a minha tia Fátima que são as minhas inspirações e as grandes responsáveis por eu ter tido acesso ao ensino superior. Agradeço também a minha irmã Helena, que é minha motivação diária.

Agradeço ao meu namorado Ícaro, por todo apoio e parceria ao longo da graduação e da vida.

Agradeço ao meu orientador Edson, pela excelente orientação, pela paciência e por me proporcionar aprendizados determinantes para minha carreira e para vida.

Resumo

O número de iniciativas que promovem competições de programação vem crescendo cada vez mais, incentivando profissionais de tecnologia na aquisição e aperfeiçoamento de habilidades técnicas. Os problemas a serem resolvidos nessas maratonas podem ter a seguinte categorização, dentre outras: Geometria Computacional, Força Bruta, AdHoc, Programação Dinâmica, String, Matemática e Grafos. O presente trabalho tem o objetivo de implementar e analisar algoritmos de Geometria Computacional utilizando a técnica de *sweep line* voltada para maratonas de programação. A análise dos algoritmos foi feita seguindo os requisitos de viabilidade de uso em competições de programação: baixa complexidade de tempo e fácil implementação. Foram analisados os seguintes algoritmos: pontos de interseção de segmentos de reta, par de pontos mais próximos, envoltório convexo de Graham e de Andrew. O trabalho mostra que as soluções apresentadas com *sweep line* são, na maioria dos casos, viáveis de serem usadas em maratonas.

Palavras-chave: *sweep line*. varredura. programação para competição. complexidade assintótica.

Abstract

The number of initiatives promoting programming competitions has been growing more and more, encouraging technology professionals to acquire and improve technical skills. The problems to be solved in these marathons can have the following categorization, among others: Computational Geometry, Brute Force, AdHoc, Dynamic Programming, String, Mathematics and Graphs. This work aims to implement and analyze computational geometry algorithms using the sweep line technique aimed at programming marathons. The algorithms' analysis was performed following the requirements of the feasibility of usage in programming competitions: low execution time complexity and easy implementation. The following algorithms were analyzed: line segments' intersection points, pair closest points, Graham's and Andrew's convex hull. The work shows that the sweep line solutions are, in most cases, feasible to be used in programming contests.

Key-words: sweep line. scan. competitive programming. asymptotic complexity.

Lista de ilustrações

Figura 1 – *Sweep Line.* 27
Figura 2 – Fluxo da *sweep line.* 28
Figura 3 – Árvore de Fenwick. 30
Figura 4 – Envoltório Convexo. 31

Figura 5 – Interseção de 10 segmentos de reta. 37
Figura 6 – Par de pontos mais próximos para um conjunto de 10 pontos. 38
Figura 7 – Polígonos - 100 pontos. 39

Figura 8 – *Sweep Line* nos pontos de interseção dos segmentos. 44
Figura 9 – Tempos de execução - Pontos de interseção de segmentos de reta. . . . 46
Figura 10 – *Sweep Line* no par de pontos mais próximos. 49
Figura 11 – Tempos de execução - Par de pontos mais próximos. 51
Figura 12 – Visualização do ângulo utilizado na ordenação do algoritmo de Graham. 52
Figura 13 – *Sweep Line* no envoltório convexo de Graham. 54
Figura 14 – Tempos de execução - Envoltório convexo de Graham. 56
Figura 15 – *Sweep Line* no envoltório convexo de Andrew. 60
Figura 16 – Tempos de execução - Envoltório convexo de Andrew. 61

Lista de tabelas

Tabela 1 – Quantidades de linhas dos códigos de exemplo. 32

Tabela 2 – Métricas de tamanho de código para o algoritmo de interseção. 45
Tabela 3 – Métricas de tamanho de código para o algoritmo de par de pontos. . . 50
Tabela 4 – Métricas de tamanho de código para o algoritmo Graham. 55
Tabela 5 – Métricas de tamanho de código para o algoritmo Andrew. 60

Lista de quadros

1.1 Complexidades mais comuns em Big-O. 29

2.1 Ferramentas Utilizadas. 36
2.2 Algoritmos escolhidos. 37

Lista de códigos

1 Exemplo em Python para contagem de linhas. 32
2 Exemplo em C++ para contagem de linhas. 32
3 Pontos de interseção de segmentos de reta - C++ 70
4 Pontos de interseção de segmentos de reta - Python 72
5 Par de pontos mais próximos - C++ 74
6 Par de pontos mais próximos - Python 76
7 Envoltório Convexo de Graham - C++ 79
8 Envoltório Convexo de Graham - Python 81
9 Envoltório Convexo de Andrew - C++ 84
10 Envoltório Convexo de Andrew - Python 86
11 Script para melhor caso do algoritmo de interseção de segmentos de reta . 88
12 Script para pior caso do algoritmo de interseção de segmentos de reta . . . 89
13 Script para melhor caso do algoritmo de par de pontos mais próximos . . . 91
14 Script para pior caso do algoritmo de par de pontos mais próximos 92
15 Script para triângulo como envoltório convexo 94
16 Script para retângulo como envoltório convexo 96
17 Script para polígono aleatório como envoltório convexo 97

Lista de abreviaturas e siglas

Capes	Coordenação de Aperfeiçoamento de Pessoal de Nível Superior
BIT	*Binary Indexed Tree*
RSQ	*Range Sum Queries*
TCC	Trabalho de Conclusão de Curso

Sumário

Introdução 25

1 FUNDAMENTAÇÃO TEÓRICA 27
1.1 *Sweep Line* 27
1.2 Complexidade assintótica de algoritmo 28
1.3 Árvore de Fenwick 29
1.4 Envoltório Convexo 30
1.5 Métricas de tamanho de código 31

2 METODOLOGIA 35
2.1 Fontes bibliográficas 35
2.2 Ferramentas utilizadas 35
2.3 Algoritmos analisados 36
2.4 Casos de teste 36

3 RESULTADOS 41
3.1 Pontos de interseção de segmentos de reta 41
3.1.1 Solução do problema com *sweep line* 41
3.1.2 Análise da complexidade 44
3.1.3 Análise da simplicidade de implementação 45
3.1.4 Comparação de tempo entre as implementações 45
3.1.5 Exemplo de problema 46
3.2 Par de pontos mais próximos 46
3.2.1 Solução do problema com *sweep line* 47
3.2.2 Análise da complexidade 47
3.2.3 Análise da simplicidade de implementação 50
3.2.4 Comparação de tempo entre as implementações 50
3.2.5 Exemplo de problema 51
3.3 Envoltório Convexo de Graham 51
3.3.1 Solução do problema com *sweep line* 51
3.3.2 Análise da complexidade 55
3.3.3 Análise da simplicidade de implementação 55
3.3.4 Comparação de tempo entre as implementações 56
3.3.5 Exemplo de problema 57
3.4 Envoltório Convexo de Andrew 57
3.4.1 Solução do problema com *sweep line* 57

3.4.2 Análise da complexidade 59
3.4.3 Análise da simplicidade de implementação 59
3.4.4 Comparação de tempo entre as implementações 60
3.4.5 Exemplo de problema 61

4 CONSIDERAÇÕES FINAIS 63
4.1 Conclusões sobre os resultados 63
4.2 Dificuldades encontradas 63
4.3 Trabalhos futuros 64

REFERÊNCIAS 65

APÊNDICE A – CÓDIGOS DO ALGORITMO DE PONTOS DE INTERSEÇÃO DE SEGMENTOS DE RETA . . . 67
A.1 C++ 67
A.2 Python 70

APÊNDICE B – CÓDIGOS DO ALGORITMO DE PAR DE PONTOS MAIS PRÓXIMOS 73
B.1 C++ 73
B.2 Python 75

APÊNDICE C – CÓDIGOS DO ALGORITMO DO ENVOLTÓRIO CONVEXO DE GRAHAM 77
C.1 C++ 77
C.2 Python 79

APÊNDICE D – CÓDIGOS DO ALGORITMO DO ENVOLTÓRIO CONVEXO DE ANDREW 83
D.1 C++ 83
D.2 Python 85

APÊNDICE E – *SCRIPTS* PARA GERAR ENTRADAS DO ALGORITMO DE INTERSEÇÃO DE SEGMENTOS DE RETA 87
E.1 Melhor caso 87
E.2 Pior caso 88

APÊNDICE F – *SCRIPTS* PARA GERAR ENTRADAS DO ALGORITMO DE PAR DE PONTOS MAIS PRÓXIMOS 91

F.1 Melhor caso 91
F.2 Pior caso 91

APÊNDICE G – *SCRIPTS* PARA GERAR ENTRADAS DOS ALGORITMOS DE ENVOLTÓRIO CONVEXO . . . 93
G.1 Triângulo 93
G.2 Retângulo 95
G.3 Polígono aleatório 96

Introdução

Atualmente há diversas iniciativas que promovem competições de programação. Uma delas é a competição internacional ACM ICPC[1] (*International Collegiate Programming Contest*), patrocinada pela IBM. No Brasil este evento recebe o nome de Maratona de Programação e equipes brasileiras participam dela desde 1996 (SBC, 2020). O objetivo é o time, composto de três competidores, resolver o máximo de problemas em cinco horas. Os problemas a serem resolvidos nessa maratona podem ter a seguinte categorização, dentre outras: Geometria Computacional, Força Bruta, *Ad Hoc*, Programação Dinâmica, String, Matemática e Grafos (HALIM; HALIM, 2013).

Para ir bem nas competições, o time precisa se preparar para conseguir resolver problemas conhecidos da computação. Os códigos não precisam ter as melhores soluções, mas precisam produzir as mesmas saídas estipuladas no problema e o tempo de execução deve estar dentro do tempo limite também estipulado pelo autor do problema (SBC, 2020). O tempo que a equipe leva para resolver um problema também influencia em sua posição no ranking, então além de resolver o problema, os maratonistas precisam implementar rapidamente as soluções (ICPC, 2020). É fundamental, então, que os competidores conheçam e saibam identificar os melhores algoritmos, dentro de cada tópico, em termos de tempo de execução e facilidade de implementação.

O *sweep line*, como uma técnica de aplicação geral, pode ajudar na formação do maratonista e na sua capacidade de solução de problemas de geometria computacional, mas ainda faltam materiais didáticos que apresentem a técnica de forma abrangente, principalmente para estudantes iniciantes.

Justificativa

Preparar-se para as maratonas de programação decorando muitos algoritmos específicos não é didático e nem prático. O ideal é que os competidores se preparem estudando técnicas abrangentes que compreendem famílias de algoritmos, como programação dinâmica, força bruta, entre outras. Assim o estudante será capaz de utilizar uma mesma técnica em problemas diferentes e compreender melhor os fundamentos dos algoritmos baseados em cada técnica.

O *sweep line* pode ser apresentado como uma destas técnicas abrangentes, fundamentando vários problemas de geometria computacional, como, por exemplo, determinar envoltório convexo de um conjunto de pontos, encontrar o par de pontos mais próximos,

[1] <https://icpc.global/>

contar pontos de interseções, entre outros. Mas há pouco material didático que aborde esta técnica em particular, sobre tudo de forma sistemática. Mas ainda não há, na literatura, materiais que façam essa apresentação de forma sistemática.

A proposta deste trabalho é apresentar a técnica de *sweep line* como um paradigma de solução de problemas geométricos, ilustrando-a por meio da exposição de diversos algoritmos da geometria computacional que são baseadas no *sweep line*, para que o estudante conheça este paradigma e o possa aplicar, seja no entendimento destes algoritmos, seja na elaboração de novos algoritmos para a solução de problemas.

Objetivos

O objetivo geral é apresentar *sweep line* como um paradigma de solução de problemas de geometria computacional. Para isso serão implementados e analisados alguns algoritmos e soluções de problemas que se enquadram no paradigma e que possam ter um desempenho e tempo de implementação adequados ao seu uso no contexto de competições de programação.

Os objetivos específicos deste trabalho são:

- identificar e implementar algoritmos que utilizam esta técnica;
- avaliar os algoritmos de acordo com requisitos de complexidade e velocidade de implementação;
- identificar quais implementações são mais apropriadas para uso em competições de programação.

Estrutura do Trabalho

Este trabalho está dividido em 5 capítulos. O Capítulo 1 expõe conceitos e fundamentos necessários para o entendimento do uso de *sweep line*. O Capítulo 2 descreve os métodos e recursos utilizados no desenvolvimento do trabalho como a estratégia de análise e critérios de escolha dos algoritmos. O Capítulo 3 apresenta as implementações com *sweep line* dos problemas analisados, expondo a viabilidade de uso dos algoritmos. Por último, no Capítulo 4 são feitas considerações finais sobre os resultados.

1 Fundamentação Teórica

Tempo de implementação e ordem de complexidade baixos são características muito importantes em algoritmos utilizados em maratonas de programação. Para entender essas características é preciso explorar alguns conceitos da Ciência da Computação.

Neste capítulo serão apresentados alguns desses conceitos e características como a definição de *sweep line* na Seção 1.1, complexidade assintótica na Seção 1.2, a estrutura de árvore de Fenwick na Seção 1.3, a definição de envoltório convexo na Seção 1.4 e por último são apresentadas métricas de tamanho de código, na Seção 1.5.

1.1 *Sweep Line*

A técnica de *sweep line* ou de *sweep plane* consiste em fazer varreduras em um plano usando uma linha de inspeção para encontrar pontos de intersecção. Essa técnica se caracteriza como um paradigma de geometria computacional, fundamentando vários algoritmos.

A *sweep line* se move em um sentido determinado e sempre que ela passa por um ponto de intersecção ela para e atualiza o seu *status*. Esses pontos são chamados pontos de parada e são processados em uma determinada ordem, seja ela implícita ou explícita. Essa ordem é quem determina o sentido em que a linha se move (KEATING, 2005). Na Figura 1 é mostrado um exemplo de *sweep line* se movimentando em um plano. Para

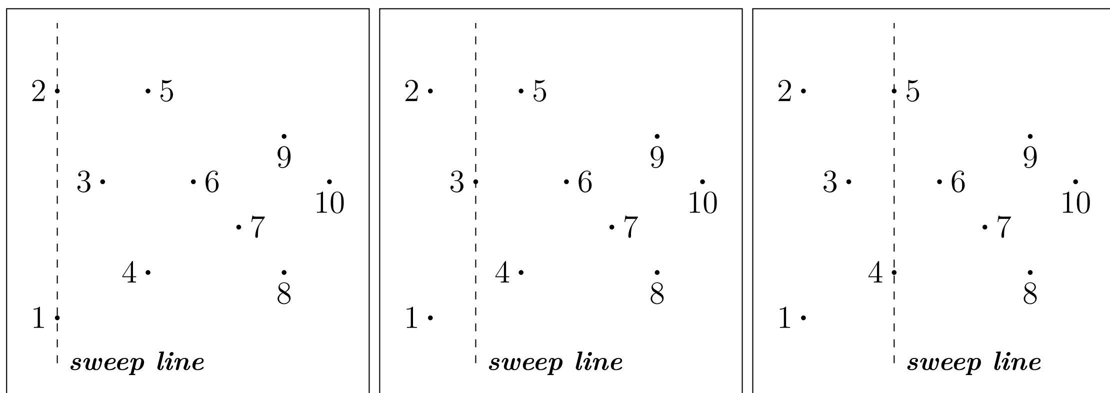


Figura 1 – *Sweep Line*.

discretizar a varredura, o conjunto de pontos pode ser interpretado como eventos que são baseados no problema que está sendo considerado. Os eventos são visitados apenas uma vez e são armazenados em estruturas de dados em que seja possível processá-los em uma ordem pré-definida. A Figura 2 mostra esse fluxo de varredura de uma *sweep line*.

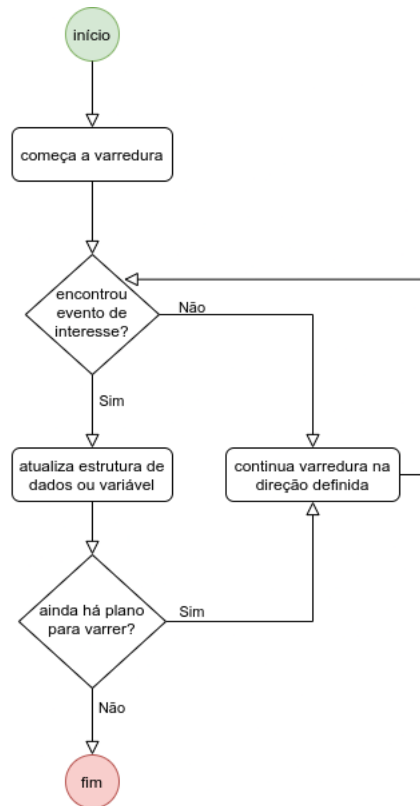


Figura 2 – Fluxo da *sweep line*.

A complexidade de um algoritmo de *sweep plane* depende do número de elementos geométricos no plano, do número de pontos de interseção e também da complexidade da rotina de ordenação. Assim, no caso geral a técnica é sensível à qualidade, e não apenas ao tamanho da entrada.

1.2 Complexidade assintótica de algoritmo

Uma análise assintótica calcula o custo de um algoritmo em termos de memória e de tempo de processamento necessários para sua execução, fornecendo indicativos sobre a eficiência do algoritmo. A complexidade assintótica pode ser usada como medida de comparação entre diferentes algoritmos, então ela tem grande importância na escolha de abordagens para uma executar uma determinada tarefa.

Crescimento assintótico é o quanto uma função cresce quando seu argumento tende ao infinito. Uma das formas utilizadas na Computação para descrever esse crescimento é a notação *Big-O* ou $O(f)$. Dada uma função f de entrada, a notação $O(f)$ denota o conjunto de todas as funções que crescem tão rápido quanto f, em seus piores casos (ZIVIANI, 1999).

Definição 1. ***(Conjunto O(f))*** *Dada duas funções* f *e* g*,* $g(n)$ *pertence ao conjunto de funções* $O(f)$ *se existem duas constantes positivas* c *e* m *tais que, para todo* $n \geq m$*,*

tem-se

$$g(n) \leq cf(n). \tag{1.1}$$

Essa notação ignora o coeficiente constante porque, no limite, f e g tem mesmo comportamento, ou seja, o limite do quociente $f(n)/g(n)$ é uma constante quando n tende ao infinito. As funções do conjunto $O(f)$ são cotas superiores, portanto, são aproximações para f.

As complexidades de alguns algoritmos são sensíveis à qualidade da entrada, o que significa que não só o tamanho da entrada influencia na complexidade mas também fatores como ordenação, discretização, entre outros.

Normalmente, os algoritmos são agrupados em classes de comportamento assintótico que determinam a complexidade inerente do algoritmo. Quando dois algoritmos fazem parte da mesma classe de comportamento assintótico, eles podem ser considerados equivalentes. O Quadro 1.1 apresenta as complexidades mais comuns e suas classes em notação *Big-O*.

Quadro 1.1 – Complexidades mais comuns em Big-O.

Notação	**Classe**
$O(1)$	constante
$O(\log n)$	logarítmica
$O(n)$	linear
$O(n \log n)$	linearítmica
$O(n^2)$	quadrática
$O(2^n)$	exponencial
$O(n!)$	fatorial

1.3 Árvore de Fenwick

Algoritmos que se baseiam na *sweep line* podem requerer o uso de estruturas de dados especiais, como é o caso do algoritmo de pontos de interseção de segmentos de reta (Seção 3.1), que para manipular seus dados de forma eficiente pode fazer uso de uma árvore de Fenwick.

A árvore de Fenwick, também chamada de *Binary Indexed Tree (BIT)*, é uma estrutura de dados que permite realizar consultas sobre a soma de elementos em um intervalo de índices (*range sum queries (RSQ)*) e atualização de valores com complexidade $O(\log N)$, onde N é o número de elementos armazenados na árvore.

A *BIT* é implementada implicitamente por meio de *array* em que o índice zero é descartado. Nesse *array* são armazenadas as somas de intervalos de índice da sequência de forma que qualquer intervalo $[1, j]$ seja representado por meio da união de $O(\log N)$ intervalos disjuntos (FENWICK, 1994).

A ideia é que, como um número inteiro é a soma das potências de 2, uma frequência cumulativa pode ser representada como a soma apropriada de conjuntos de subfrequências cumulativas, como na Figura 3.

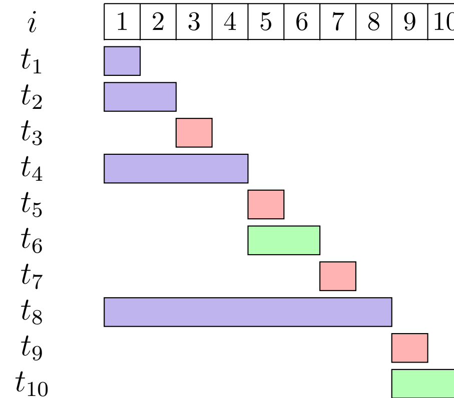


Figura 3 – Árvore de Fenwick.

Seja t o *array* da árvore e $p(n)$ a maior potência de 2 que divide o inteiro positivo n, então, t_i será a soma dos elementos de a_k cujos índices pertencem ao intervalo $I_i = [i - p(i) + 1, i]$, ou seja,

$$t_i = \sum_{k=i-p(i)+1}^{i} a_k. \tag{1.2}$$

No cálculo da *RSQ*, um novo índice é obtido retirando o *bit* menos significativo do índice antigo e repetindo essa operação até que o índice seja zero.

Uma implementação em C++ da árvore de Fenwick está apresentada em A.1, a qual foi obtida do repositório[1] de materiais da disciplina de Tópicos Especiais em Programação do curso de Engenharia de Software.

1.4 Envoltório Convexo

Alguns algoritmos que determinam o envoltório convexo de um conjunto usam *sweep line*, como é o caso do algoritmo de Graham (Seção 3.3) e da cadeia monótona de Andrew (Seção 3.4).

Dado um conjunto S de N pontos, uma região é dita convexa se, para cada par de pontos no interior da região, cada ponto em um segmento de reta que une esse par também está completamente contido no interior. O envoltório convexo $C_H(S)$ é o menor polígono convexo que contém todos os pontos de S e cujos vértices também são pontos de S (LAAKSONEN, 2018).

1 <https://github.com/edsomjr/TEP>

O objetivo dos algoritmos de envoltório convexo é determinar se cada ponto de S pertence ao $C_H(S)$. Para os pontos da Figura 4a, por exemplo, o envoltório convexo é a Figura 4b.

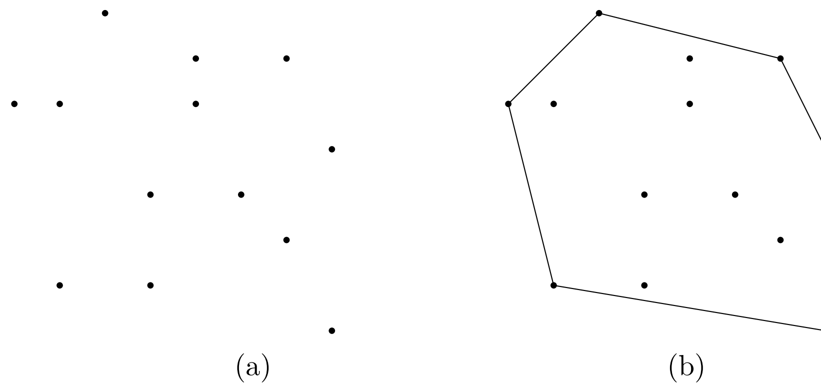


Figura 4 – Envoltório Convexo.

1.5 Métricas de tamanho de código

Existem algumas métricas que ajudam a calcular o tamanho de um código. Esse tamanho pode influenciar no quanto o código é fácil de memorizar e no quanto seu uso é viável em competições de programação. A maioria das linguagens de programação permite uma variedade de estilos de escrita que provoca variações na quantidade de linhas de código, portanto há algumas métricas que podem ser consideradas como a quantidade de linhas de código.

A métrica linhas de código conta todas as linhas que contêm código, excluindo linhas de comentários e linhas em branco. Já linhas físicas de código conta o número total de linhas que tem o arquivo de código, incluindo comentários e linhas em branco.

A quantidade de linhas lógicas de código diz o número de instruções realizadas. Na linguagem C, por exemplo, pode ser obtida contando o número de pontos-e-vírgulas (;). Então, linhas com condicionais e declaração de funções não são contadas. Linhas com mais de uma instrução são contadas mais de uma vez. Por fim, linhas efetivas de código conta todas as linhas que contêm código e que não tenham apenas delimitadores, como chaves, parênteses, aspas, etc.

Os Códigos 1 e 2 possuem as mesmas funcionalidades mas estão escritos em linguagens de programação diferentes: o primeiro em Python 3.7 e o segundo em C++17. Eles serão usados para exemplificar a contagem de quantidade de linhas na Tabela 1.

```
1 def get_name():
2     print("What's your name?")
3     name = input()
4
```

```
5        return name
6
7    def show_name(name):
8        print(f"Hello, {name}!")
9
10   name = get_name()
11   show_name(name)
```

Código 1 – Exemplo em Python para contagem de linhas.

```
1    #include <bits/stdc++.h>
2
3    using namespace std;
4
5    string get_name(){
6        string name;
7
8        cout << "What's your name?\n";
9        cin >> name;
10
11       return name;
12   }
13
14   void show_name(string name){
15       cout << "Hello, " << name << "!\n";
16   }
17
18   int main(){
19       string name;
20
21       name = get_name();
22       show_name(name);
23
24       return 0;
25   }
```

Código 2 – Exemplo em C++ para contagem de linhas.

Tabela 1 – Quantidades de linhas dos códigos de exemplo.

Métrica	**C++**	**Python**
linhas de código	17	8
linhas físicas de código	25	11
linhas lógicas de código	10	6
linhas efetivas de código	14	8

A quantidade de linhas de código pode ser usada para medir produtividade, mas não necessariamente significa mais funcionalidades ou que está sendo implementada a

melhor solução. Mas em programação competitiva uma implementação com um menor número de linhas é mais fácil de memorizar, de se registrar nas notas impressas que podem ser levadas para a competição e também de se digitar, obtendo-se assim uma vantagem de tempo na competição.

2 Metodologia

Este capítulo apresenta a metodologia utilizada para a implementação e análise dos algoritmos do paradigma de *sweep line*. A Seção 2.1 explica como as fontes bibliográficas e referências foram buscadas. A Seção 2.2 apresenta as ferramentas usadas no trabalho. A Seção 2.3 descreve qual metodologia foi usada para analisar os algoritmos escolhidos. A Seção 2.4 explica as entradas geradas para avaliar o desempenho dos algoritmos.

2.1 Fontes bibliográficas

Para buscar fontes bibliográficas foi o utilizado o Portal de Periódicos Capes[1] (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) com a seguinte *string* de busca: *sweep AND (line OR plane)*. Não foram encontrados muitas fontes adequadas para o trabalho, uma vez que a maioria dos resultados aplicava *sweep line* em contextos específicos. O único resultado selecionado para ser citado neste trabalho foi o de Elmasry e Kammer (2015). Ele foi escolhido por ter como foco principal o uso de *sweep line* em diversos algoritmos, analisando de forma objetiva as complexidades e operações de cada solução.

Um livro bastante usado para consultas foi o de Laaksonen (2018). Ele serviu muito como referência para as definições e visualizações dos problemas. Ele é uma fonte adequada para este trabalho pois foi feito especificamente para competidores de programação, reunindo vários algoritmos e problemas que costumam aparecer em maratonas.

Para as visualizações, implementações e definições também foi muito usado como referência o repositório[2] que contém o material da disciplina de Tópicos Especiais de Programação, da graduação em Engenharia de Software da Universidade de Brasília. A disciplina aborda programação competitiva tratando dos principais conceitos, problemas e soluções envolvidos nas maratonas de programação tradicionais. A disciplina é ministrada pelo professor Dr. Edson Alves da Costa Júnior que também é o principal contribuidor do repositório.

2.2 Ferramentas utilizadas

As ferramentas utilizadas para o desenvolvimento do trabalho estão no Quadro 2.1. Tradicionalmente, as linguagens mais utilizadas em maratonas de programação são Python e C/C++, por esse motivo essas linguagens foram escolhidas para o trabalho.

[1] <http://www.periodicos.capes.gov.br/>
[2] <https://github.com/edsomjr/TEP>

Assim, o sistema operacional utilizado em maratonas deve ser capaz de compilar e executar os códigos desenvolvidos em C/C++ com compiladores `gcc/g++`, os quais também foram utilizados neste trabalho.

Quadro 2.1 – Ferramentas Utilizadas.

	Ferramentas
notebook	Lenovo Ideapead 320 (Intel i5-7200U, 2GB de RAM)
sistema operacional	MX Linux 19
editor	NeoVim 0.3.4
linguagens de programação	C++17
	Python 3.7
compilador C++	g++ 8.3.0-6

2.3 Algoritmos analisados

A escolha dos algoritmos foi baseada no objetivo do trabalho que é apresentar *sweep line* como um paradigma de solução de problemas que eventualmente apareçam em maratonas de programação. Então foram escolhidos algoritmos que resolvem problemas clássicos da geometria computacional e que sejam candidatos a serem usados em maratonas.

A viabilidade de uso do algoritmo em uma maratona dá-se pela eficiência e pela rapidez de implementação, assim o algoritmo precisa satisfazer os requisitos de baixa ordem de complexidade assintótica, para finalizar sua execução dentro do tempo estipulado, e de simplicidade de implementação, para que possam ser escritos em tempo razoável.

O primeiro requisito será analisado de acordo com a função de cada algoritmo e para o segundo serão considerados fatores como quantidade de linhas e conhecimentos que são necessários para implementação da solução, como estruturas de dados especiais, algoritmos de busca, entre outros.

O desenvolvimento do trabalho foi dividido em duas partes: a primeira corresponde à disciplina de Trabalho de Conclusão de Curso 1 (TCC 1) e a segunda ao Trabalho de Conclusão de Curso 2 (TCC 2). Os algoritmos escolhidos foram analisados em momentos diferentes, alguns durante o TCC 1 e outros durante o TCC 2. O Quadro 2.2 lista esse algoritmos.

2.4 Casos de teste

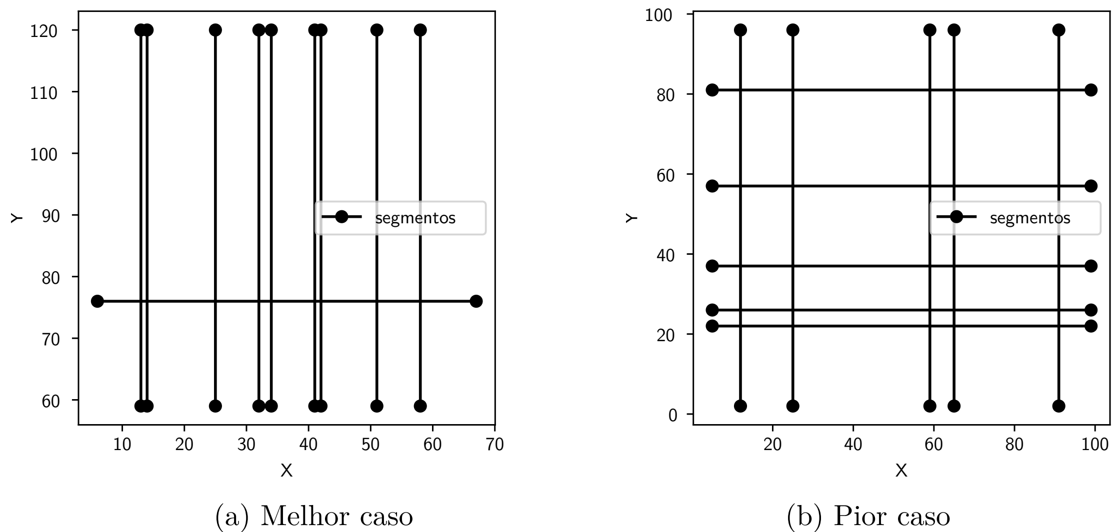
Para comparação de tempo de execução entre as implementações em C++ e as implementações em Python, foram escritos *scripts* em Python para gerar entradas com

Quadro 2.2 – Algoritmos escolhidos.

	Algoritmos	**Linguagens**
TCC 1	pontos de interseção de segmentos de reta	C++
	par de pontos mais próximos	C++
TCC 2	pontos de interseção de segmentos de reta	Python
	par de pontos mais próximos	Python
	envoltório convexo de Graham	C++ e Python
	envoltório convexo de Andrew	C++ e Python

os melhores e piores casos de cada algoritmo. Para cada tipo de entrada, os tamanhos dos conjuntos de elementos são as 6 primeiras potências de dez: $10, 10^2, 10^3, 10^4, 10^5$ e 10^6.

Para o algoritmo de contagem de pontos de interseção entre segmentos de reta paralelos aos eixos ortogonais, um melhor caso não trivial acontece quando, para N segmentos, $N - 1$ deles estão posicionados horizontalmente, e 1 está posicionado verticalmente, interceptando todos os segmentos horizontais, totalizando $N - 1$ pontos de interseção. Na Figura 5a está representado esse caso para $N = 10$. O pior caso acontece quando há $N/2$ segmentos horizontais e $N/2$ segmentos verticais e todos eles se cruzam, totalizando $(N/2)^2$ interseções, conforme mostrado na Figura 5b, para $N = 10$. Os *scripts* utilizados para gerar estas entradas desse algoritmo estão no Apêndice E.



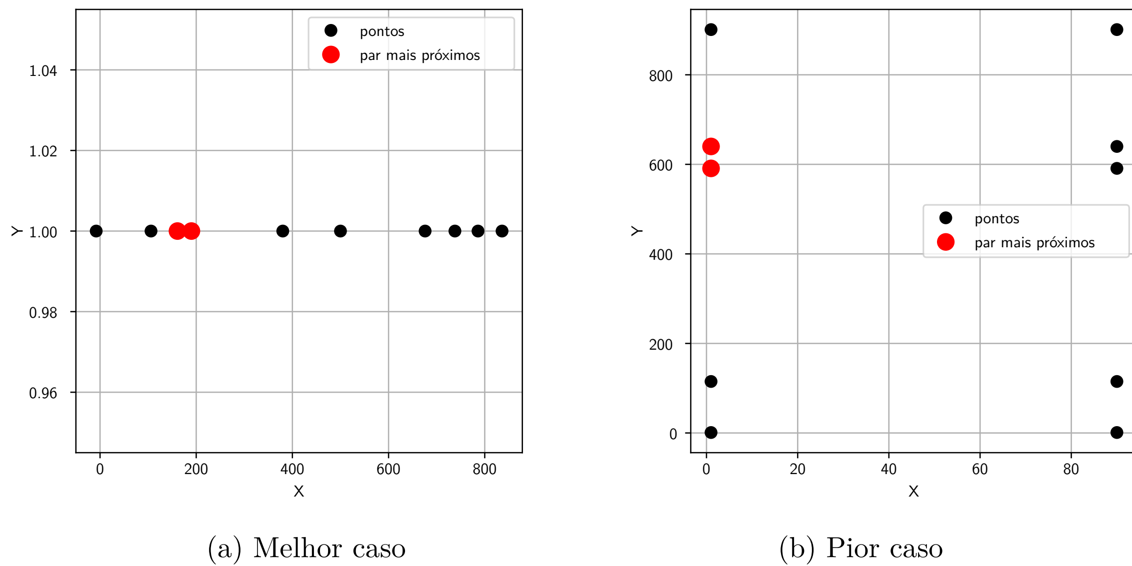
(a) Melhor caso

(b) Pior caso

Figura 5 – Interseção de 10 segmentos de reta.

Para o algoritmo de par de pontos mais próximos, um melhor caso não trivial acontece quando todos os N pontos estão na mesma reta, como é mostrado na Figura 6a, para $N = 10$. O pior caso é quando os N pontos estão distribuídos na mesma quantidade e alinhados em duas retas, como demonstra a Figura 6b, para $N = 10$. Os *scripts* para gerar estas entradas estão no Apêndice F.

Para os algoritmos do envoltório convexo de Andrew e de Graham, 3 tipos de

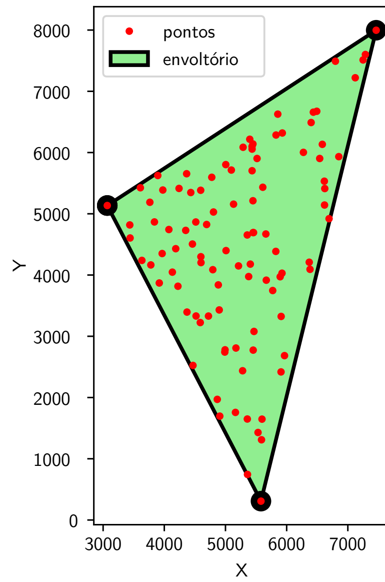


(a) Melhor caso (b) Pior caso

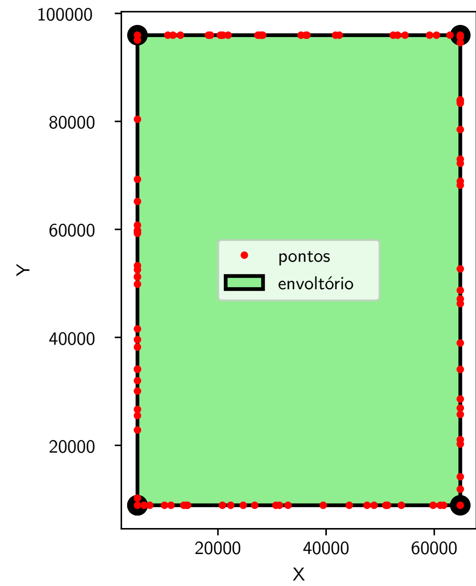
Figura 6 – Par de pontos mais próximos para um conjunto de 10 pontos.

entradas foram geradas. No primeiro, o envoltório convexo forma um triângulo, sendo o melhor caso de entrada, pois apenas 3 pontos fazem parte do envoltório. A Figura 7a mostra um exemplo desse tipo de entrada para $N = 100$. No segundo tipo de entrada, o envoltório forma um retângulo em que todos os pontos do envoltório fazem parte desse retângulo, sendo assim o pior caso. A Figura 7b exemplifica esse retângulo para quantidade de pontos $N = 100$. No terceiro tipo de entrada, o polígono formado pelo envoltório convexo é aleatório. Na Figura 7c está um exemplo de polígono aleatório. Os *scripts* para fazer a geração destas entradas estão no Apêndice G.

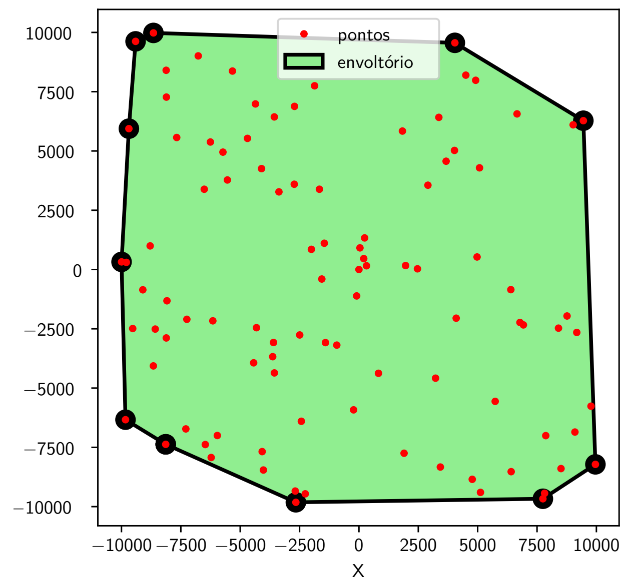
Os tempos foram mensurados a partir somente da execução dos algoritmos, desconsiderando os tempos de leitura da entrada e de escrita da saída. Para os tempos em C++, foi utilizada a biblioteca *chrono* e em Python foi utilizada a biblioteca *time*. Com ambas, foi usado o melhor *clock* disponível: `high_resolution_clock` e `perf_counter()`, respectivamente.



(a) Triângulo



(b) Retângulo



(c) Aleatório

Figura 7 – Polígonos - 100 pontos.

3 Resultados

Neste capítulo são implementadas e analisadas soluções para problemas em que pode ser identificada a *sweep line*. Na Seção 3.1 é explorado o problema de interseção de segmentos de reta, já na Seção 3.2 é apresentado o problema de par de pontos mais próximos. Por fim, na Seção 3.3 e na Seção 3.4 são apresentados algoritmos que encontram o envoltório convexo de um conjunto de pontos.

3.1 Pontos de interseção de segmentos de reta

Dado um conjunto S de N segmentos de reta, sendo eles horizontais ou verticais, o problema é contar o número total de pontos de interseção entre esses segmentos. Um ponto é de interseção se ele está simultaneamente em um segmento vertical e em um horizontal e, neste problema, é assumido que os segmentos não possuem interseção com outros na mesma direção (LAAKSONEN, 2018).

O problema pode ser resolvido em $O(N^2)$ com um algoritmo de força bruta que compara cada segmento a todos os outros, passando por todos os pares possíveis e verificando se eles se cruzam. Mas usando *sweep line* e uma estrutura de dados para consulta de intervalo o problema pode ser resolvido de forma mais eficiente.

3.1.1 Solução do problema com *sweep line*

A solução com *sweep line* consiste em processar os pontos das extremidades dos segmentos da esquerda para a direita observando três tipos de eventos:

1. início de um intervalo horizontal com altura y,
2. intervalo vertical $[a, b]$,
3. fim de um intervalo horizontal com altura y.

O conjunto de pontos dos intervalos dos segmentos deve estar ordenado primeiro por coordenada x, segundo por coordenada y. Primeiro é preciso percorrer os intervalos e identificar os eventos, que devem ser ordenados por coordenada x, em seguida por seu tipo e por último, pelo índice do respectivo intervalo.

É preciso uma estrutura de dados para armazenar um conjunto de coordenadas y onde há um segmento horizontal. A linha de varredura percorre o conjunto de eventos e quando encontra o evento 1, a coordenada y é a adicionada ao conjunto; quando encontra o

evento 3, a coordenada y é removida do conjunto. A estrutura usada para as coordenadas y deve ser uma árvore de indexação binária, também chamada de árvore de Fenwick (Seção 1.3).

Quando há um segmento vertical de intervalo $[a, b]$, o qual é representado por um evento do tipo 2, é contabilizado o número de segmentos horizontais cuja coordenada y está dentro deste intervalo e esse número é incrementado ao número total de pontos de interseção.

A solução está representada em pseudo-código no Algoritmo 1.

Algorithm 1 Pontos de interseção de segmentos de retas horizontais e verticais.

Entradas: *intervalos* = lista com os *intervalos* de coordenadas; *ft* = BIT com RSQ.

Saída: *total* = número total de pontos de interseção.

```
 1: total ← 0
 2: eventos ← []
 3:
 4: "Ordene os intervalos por coordenadas x e y"
 5: intervalos ← sort_by_x_and_y(intervalos)
 6:
 7: for i : intervalos do
 8:     if i.A.x == I.B.x then
 9:         eventos.add(2, xmin, i)
10:     else
11:         eventos.add(1, xmin, i)
12:         eventos.add(3, xmax, i)
13:     end if
14: end for
15:
16: "Ordene os eventos por coordenada x, tipo e por índice i"
17: eventos ← sort_by_x_type_index(P)
18:
19: for e : eventos do
20:     for A, B : intervalos do
21:         if e.tipo = 1 then
22:             ft.add(A.y)
23:         else if e.tipo = 2 then
24:             ymin ← min(A.y, B.y)
25:             ymax ← max(A.y, B.y)
26:             total ← total + ft.RSQ(ymin, ymax)
27:         else
28:             ft.remove(B.y)
29:         end if
30:     end for
31: end for
32:
33: return total
```

A *sweep line* passa pelos eventos e verifica o tipo de cada um, se for do tipo 1, a coordenada y do ponto inicial do segmento é adicionada na árvore de Fenwick, mas se o evento for do tipo 2, o número total é incrementado com a RSQ da maior e menor coordenada y do segmento. Caso o evento seja do tipo 3, a coordenada y final é decrementada da árvore. A implementação em C++ do Algoritmo 1 está apresentada na Seção A.1 e a implementação em Python está apresentada em A.2.

A ordenação subjacente dos intervalos em C++ foi feita inserindo os elementos em um *set*, estrutura de dados cuja inserção padrão é feita em ordem crescente. Os *sets* são implementados como árvores binárias de busca auto balanceadas. Para ordenar os eventos foi preciso definir o operador `<` na estrutura do evento para que a coordenada x fosse priorizada, seguida do tipo e do índice.

Na implementação em Python, os elementos também foram armazenados na estrutura de *set* da linguagem, mas como nela não há inserção ordenada, foi preciso utilizar a função `sorted()`. Essa função usa o TimSort, que é um algoritmo de classificação baseado em *Insertion Sort* e *Merge Sort* (GEEKSFORGEEKS, 2021b).

Na Figura 8 os pontos em azul representam eventos do tipo 1, em verde são eventos do tipo 2 e em vermelho são eventos do tipo 3. Nela é possível ver os momentos em que o número total de pontos de interseção é atualizado quando a *sweep line* passa por um evento do tipo 2.

Na Figura 8b a *sweep line* começa a varredura pelo plano mas só encontra eventos do tipo 1, o início de um segmento horizontal, então as coordenadas y desses eventos são adicionadas ao conjunto. Na Figura 8c a *sweep line* passa em evento do tipo 1 e em um evento do tipo 2, um segmento vertical, então contabiliza quantas coordenadas do conjunto estão dentro do intervalo de coordenadas desse segmento, encontrando a primeira interseção, assim o valor total de pontos de interseção começa a ser incrementando em 1. Na Figura 8d a linha passa por um evento do tipo 1 e mais uma coordenada é adicionada ao conjunto.

Na Figura 8e há mais um evento do tipo 2, mas dessa vez há 3 coordenadas no conjunto que estão entre o segmento vertical, adicionando 3 ao valor total. Nesse momento a linha também passa por um evento do tipo 3, o fim de um segmento vertical, removendo a respectiva coordenada y do conjunto. Na Figura 8f são encontrados eventos do tipo 1 e 3 cujas coordenadas y são adicionas e removidas do conjunto, respectivamente. Na Figura 8g a *sweep line* passa pelo último evento do tipo 2, encontrando duas interseções, atualizando valor total para 6, além de passar também por um evento do tipo 3. Nas Figura 8h e Figura 8i a varredura continua encontrando apenas eventos do tipo 3, fazendo a operação de remoção das respectivas coordenadas do conjunto.

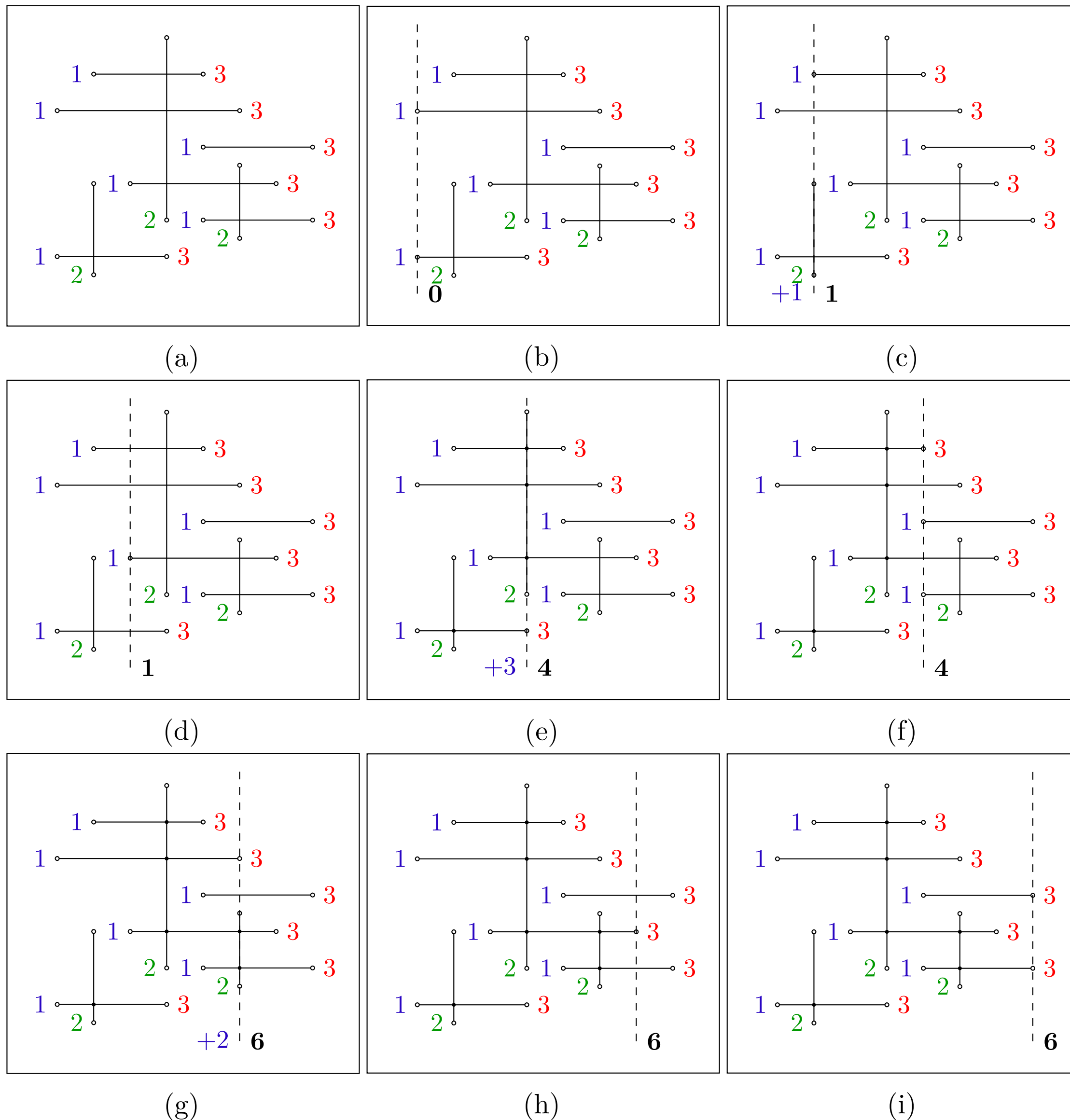


Figura 8 – *Sweep Line* nos pontos de interseção dos segmentos.

3.1.2 Análise da complexidade

Considerando que cada segmento gera no máximo 2 eventos, há no máximo $O(N)$ eventos. A ordenação dos pontos custa $O(N \log N)$ e, finalmente, há no máximo $O(N)$ eventos do tipo 2, os quais demandam uma *RSQ* que custa $O(\log N)$. Portanto, a complexidade de pior caso do algoritmo é $O(N \log N)$.

O uso de *sweep line* para resolver o problema apresentou uma ordem de complexidade menor que um algoritmo de força bruta, cuja classe assintótica é quadrática, sendo assim uma melhor alternativa.

3.1.3 Análise da simplicidade de implementação

Os tamanhos das implementações em Python e C++ estão demonstrados na Tabela 2 com as métricas descritas na Seção 1.5. Para a contagem, só estão sendo consideradas as linhas que fazem parte da implementação do algoritmo: linhas 11 a 126, na implementação em C++; linhas 7 a 78, na implementação em Python.

Pela quantidade alta de linhas físicas de código e de linhas de código em C++, o algoritmo é grande para ser implementado rapidamente. Mas em Python essas métricas ficam menores, pelo fato da linguagem ser menos verbosa e ter algumas facilidades. As quantidades de linhas efetivas de código podem variar conforme estilo de programar do competidor que, por exemplo, em C++, pode ter preferência por sempre fechar e abrir escopo em linhas separadas. As quantidades de linhas lógicas de código mostram que o algoritmo não é de fácil memorização, visto que muitas coisas precisam ser feitas nele.

Para implementar esse algoritmo é preciso que o competidor tenha conhecimento da implementação da árvore de Fenwick, que não é trivial. Esse fato aliado à quantidade alta de número de linhas faz com que o algoritmo não seja simples e nem rápido de implementar em C++, mas em Python, a implementação pode ser mais rápida.

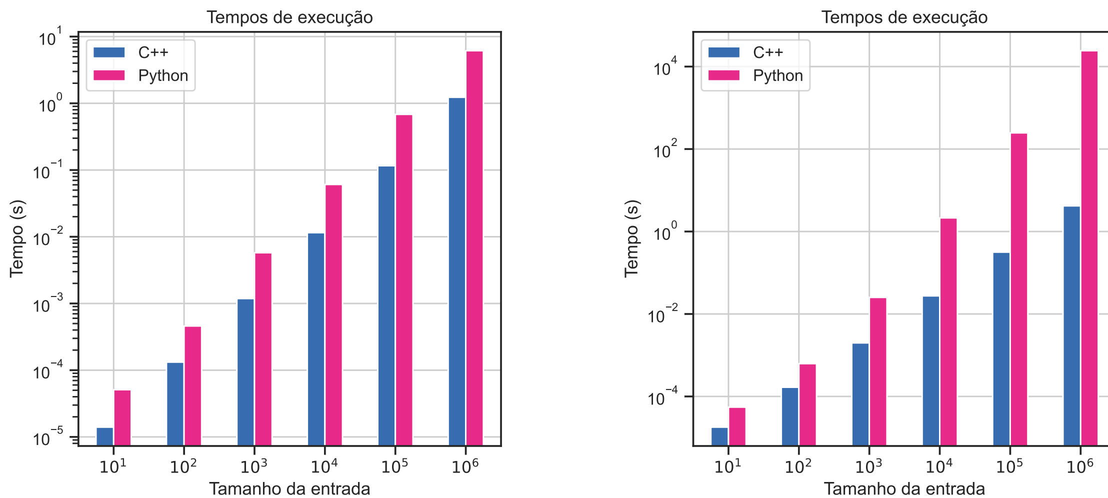
Tabela 2 – Métricas de tamanho de código para o algoritmo de interseção.

Métrica	**C++**	**Python**
linhas de código	89	53
linhas físicas de código	116	72
linhas lógicas de código	50	39
linhas efetivas de código	70	53

3.1.4 Comparação de tempo entre as implementações

A Figura 9 mostra as comparações de tempos, em escala logarítmica, das execuções em C++ e em Python para o melhor caso e o pior caso de entrada, conforme descrito na Seção 2.4. Para todos os casos e tamanhos de entrada, a execução em Python levou mais tempo. Para a entrada de 10^6 pontos do pior caso, o tempo em Python foi notavelmente maior, cerca de 5800 vezes superior ao tempo em C++ (Figura 9b).

Para o contexto de competições de programação, é inviável usar a implementação em Python, proposta por este trabalho, para problemas com entradas de tamanho 10^6 ou superiores.



(a) Tempos para melhor caso (b) Tempos para pior caso

Figura 9 – Tempos de execução - Pontos de interseção de segmentos de reta.

3.1.5 Exemplo de problema

O problema *593B - Anton and Lines* do Codeforces[1] é um exemplo de como os algoritmos de interseção de segmentos de reta podem ser usados.

Seja N $(2 \leq N \leq 10^5)$ a quantidade de segmentos que pode vir na entrada do problema, uma busca completa verificando todos os pares de segmentos, tem complexidade $O(N^2)$, o que leva a exceder o limite de tempo de 1 segundo. Mas com *sweep line* também é possível encontrar as possíveis interseções.

Para encontrar interseções no intervalo (x_1, x_2), os segmentos devem ser ordenados, em ordem crescente, pelo valor da coordenada y do segmento no ponto x_1 e, em caso de empate, pela coordena y no ponto x_2. Assim, cada segmento é processado apenas uma vez. A maior coordenada y do segundo ponto encontrada, deve ser registrada em uma variável, que inicialmente tem o valor igual a $-\infty$. A *sweep line* passa pelas coordenadas y em x_2 de cada segmento a ser processado, e se for menor do que a maior já encontrada, significa que houve uma interseção com algum dos segmentos já processados.

3.2 Par de pontos mais próximos

Dado um conjunto S de N pontos, o problema consiste em encontrar um par de pontos cuja distância euclidiana, quando comparada às distâncias entre todos os pares de pontos distintos possíveis, é mínima. O problema pode ser resolvido por meio de um algoritmo de busca completa, calculando as distâncias entre todos os pares de pontos possíveis, mas essa abordagem tem complexidade $O(N^2)$. Usando abordagem de *sweep*

[1] <https://codeforces.com/problemset/problem/593/B>

line o problema pode ser resolvido em $O(N \log N)$ (LAAKSONEN, 2018).

3.2.1 Solução do problema com *sweep line*

Sejam dois pontos P_i e $P_j \in S$ com $i \neq j$ e $d = \text{dist}(P_i, P_j)$. A ideia consiste em computar todos os vizinhos P_j de P_i cujas coordenadas x estejam no intervalo $[x - d, x]$ e coordenadas y no intervalo $[y - d, y + d]$. Essa área de vizinhança é semelhante a um retângulo cujo ponto de origem é P_i, como o mostrado na Figura 10b, em que o ponto de origem é o 3.

Os pontos vizinhos podem ser identificados mantendo um conjunto de pontos cujas coordenadas x estejam entre $[x - d, x]$, ordenado em ordem crescente de suas coordenadas y. Caso a distância de P_i para algum destes pontos seja inferior a d, o valor de d é atualizado e a varredura continua com este novo valor (ELMASRY; KAMMER, 2015). Ou seja, se tiver um ponto no interior do retângulo que fornece uma menor distância, essa nova distância será atualizada como sendo a menor, restringindo ainda mais o tamanho do retângulo nas próximas iterações.

A solução está representada em pseudo-código no Algoritmo 2. A implementação em C++ está apresentada em B.1 e a implementação em Python está apresentada em B.2.

A ordenação subjacente dos pontos em C++ foi feita com `sort()`, uma função que classifica em ordem crescente os elementos do intervalo passado como parâmetro para a função. O algoritmo de ordenação usado nessa função é o *IntroSort*, um algoritmo de classificação híbrido que usa outros três para minimizar o tempo de execução: *Quicksort*, *Heapsort* e *Insertion Sort*. O *IntroSort* varia o uso desses três dependendo do tamanho da entrada (GEEKSFORGEEKS, 2021a).

Já em Python, a ordenação foi feita com a função `sorted()` da linguagem. Essa função usa o TimSort, que é um algoritmo de classificação baseado em *Insertion Sort* e *Merge Sort* (GEEKSFORGEEKS, 2021b).

Na Figura 10 é possível ver o retângulo para os vizinhos de cada ponto. A *sweep line* é a aresta direita do retângulo, e é nela que está posicionado o ponto inspecionado atualmente. A distância inicial é a dos 2 primeiros pontos (Figura 10a). Na Figura 10b uma distância menor é encontrada então o tamanho do retângulo já é atualizado com essa nova distância na Figura 10c. Assim, a *sweep line* segue até encontrar o par de pontos com a menor distância, formado pelos pontos 6 e 3 (Figura 10g).

3.2.2 Análise da complexidade

A eficiência do algoritmo é baseada na ordenação e no fato de que a região do retângulo contém $O(1)$ pontos no caso médio.

Algorithm 2 Par de pontos mais próximos.

Entradas: P = lista de *pontos*;
Saída: *mais_proximo* = o par de pontos mais próximos.

```
1: N ← P.size()
2:
3: "Ordene os pontos P por coordenadas x e y"
4: P ← sort_by_x_and_y(P)
5:
6: mais_proximo ← (P[0], P[1])
7: distancia ← dist(P[0], P[1])
8:
9: S ← set()
10: S.add((P[0].y, P[0].x))
11: S.add((P[1].y, P[1].x))
12:
13: for p : P do
14:     for q : S do
15:         if q.x < p.x − distancia then
16:             S.remove(q)
17:             continue
18:         end if
19:
20:         if q.y > p.y + distancia then
21:             break
22:         end if
23:
24:         distancia_atual ← dist(p, q)
25:         if distancia_atual < distancia then
26:             distancia ← distancia_atual
27:             mais_proximo ← (p, q)
28:         end if
29:     end for
30:
31:     S.add((p.y, p.x))
32: end for
33:
34: return mais_proximo
```

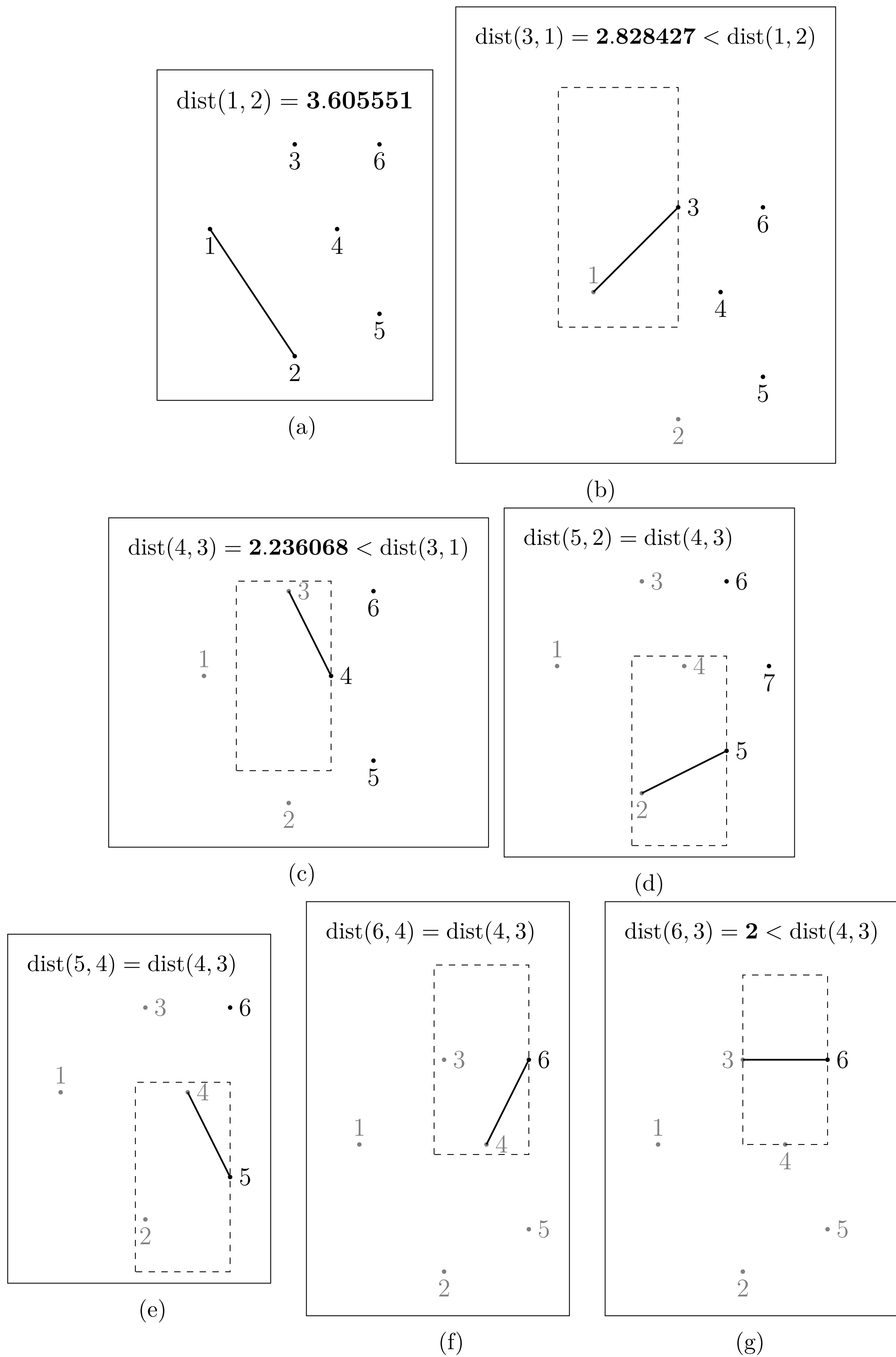


Figura 10 – *Sweep Line* no par de pontos mais próximos.

No laço de repetição que percorre os pontos, são removidos os pontos à esquerda do ponto atual cuja distância da coordenada x é maior que d. Como há N elementos no conjunto S, esse laço é executado em $O(N)$ e a complexidade de apagar o ponto do conjunto é $O(\log N)$. Então, a complexidade geral do algoritmo é $O(N \log N)$.

Portanto, o uso de *sweep line* para resolver o problema apresentou uma ordem de complexidade menor que um algoritmo de busca completa, cuja classe assintótica é quadrática.

3.2.3 Análise da simplicidade de implementação

Os tamanhos das implementações em Python e C++ estão demonstrados na Tabela 3 com as métricas descritas na Seção 1.5. Para a contagem, só estão sendo consideradas as linhas que fazem parte da implementação do algoritmo: linhas 15 a 60, na implementação em C++; linhas 12 a 56, na implementação em Python.

Pelas quantidades de linhas físicas de código o algoritmo parece ser grande. Mas as quantidades de linhas de código e de linhas efetivas de código mostram que ele não é tão grande, o que significa que pode ser implementado em um tempo razoável. As quantidades de linhas lógicas de código dizem que o número de passos não é grande o bastante que dificulte a memorização do algoritmo.

Para implementar esse algoritmo é preciso que o competidor tenha conhecimento do cálculo da distância euclidiana, mas isso faz parte dos conhecimentos elementares de matemática, não sendo necessário investir muito esforço para adquiri-lo.

Entendendo a solução do problema com *sweep line* e conhecendo as estruturas de dados básicas do C++ e do Python, a implementação do algoritmo pode ser considerada simples.

Tabela 3 – Métricas de tamanho de código para o algoritmo de par de pontos.

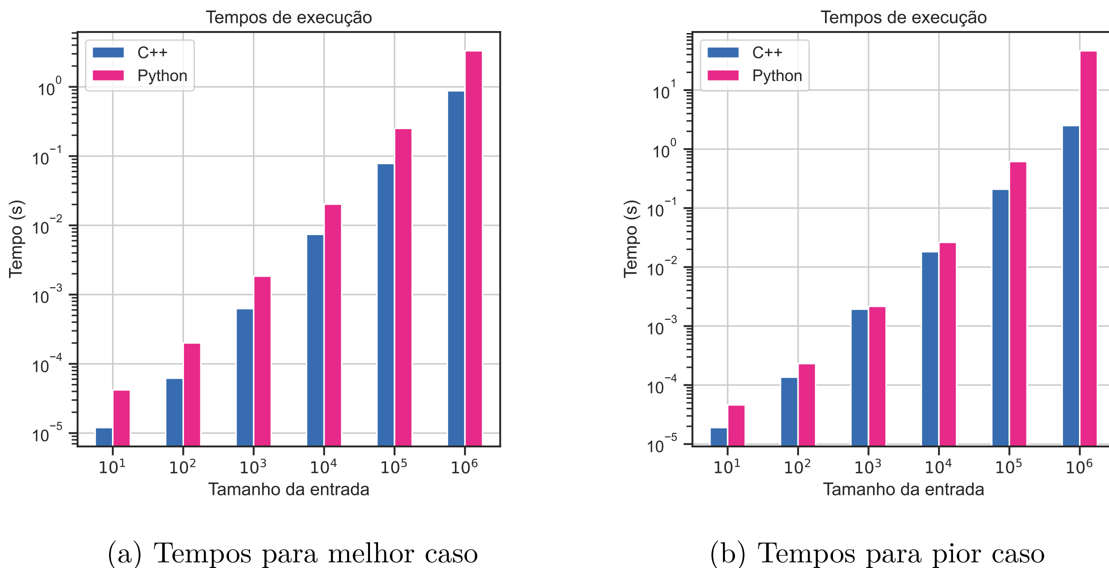
Métrica	**C++**	**Python**
linhas de código	33	31
linhas físicas de código	46	45
linhas lógicas de código	22	25
linhas efetivas de código	27	31

3.2.4 Comparação de tempo entre as implementações

A Figura 11 mostra as comparações de tempos, em escala logarítmica, das execuções em C++ e em Python para o melhor caso e o pior caso de entrada, conforme descrito na Seção 2.4. Como Python é uma linguagem interpretada e C++ uma linguagem compilada, é esperado que os tempos em Python sejam superiores aos tempos em C++. Isso

aconteceu para todos os casos e tamanhos de entrada. Confirmando esta expectativa, para a entrada de 10^6 pontos do pior caso, o tempo em Python foi notavelmente maior, cerca de 23 vezes superior ao tempo em C++ (Figura 11b).

Para o contexto de competições de programação, é inviável usar a implementação proposta em Python, para problemas com entradas de tamanho 10^6 ou superiores.



(a) Tempos para melhor caso (b) Tempos para pior caso

Figura 11 – Tempos de execução - Par de pontos mais próximos.

3.2.5 Exemplo de problema

O problema *10245 - The Closest Pair Problem* do Online Judge[2] é o problema clássico do algoritmo de par de pontos mais próximos.

A solução de *sweep line* para o algoritmo de par de pontos mais próximos pode ser submetida pra esse problema, resolvendo cada caso de teste em $O(N \log N)$, com atenção para o fato de que a entrada admite coordenadas em ponto flutuante.

3.3 Envoltório Convexo de Graham

O algoritmo do envoltório convexo de Graham foi proposto pelo matemático Ronald Graham em 1972 (GRAHAM, 1972). O algoritmo é para determinar os pontos do envoltório convexo $C_H(S)$, conforme Seção 1.4, com uma complexidade $O(N \log N)$.

3.3.1 Solução do problema com *sweep line*

Dado um conjunto finito S de N pontos, o problema é determinar o envoltório convexo $C_H(S)$. A ordenação é feita a partir de um ponto fixo, o pivô. O critério para

2 <https://cutt.ly/7RPN8Tt>

escolhê-lo é ser o ponto com a menor coordenada y e em caso de empate, o ponto com maior coordenada x. A ordem em que os pontos serão analisados é definida, de forma crescente, pelo valor do ângulo que é formado entre o ponto e o eixo x positivo do pivô, como mostra a Figura 12. Nela, o ponto pivô está na origem do plano e Θ é o ângulo formado. Assim, o ponto $P1$ será analisado pela *sweep line* antes do $P2$, por formar um ângulo menor com o pivô. Quando dois pontos formam um ângulo igual, a prioridade é dada ao que está mais perto do pivô. O ângulo pode ser obtido com arco tangente entre cada ponto e o pivô.

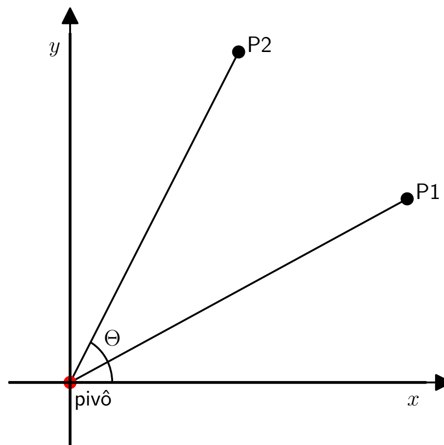


Figura 12 – Visualização do ângulo utilizado na ordenação do algoritmo de Graham.

Os pontos ordenados podem ser armazenados em um vetor v com o pivô ocupando a primeira posição. O $C_H(S)$ pode ser entendido como uma pilha e contém pelo menos 3 pontos, os quais são: o anterior ao pivô ou o último do vetor ordenado ($v[N-1]$), o pivô ($v[0]$) e o sucessor do pivô ($v[1]$). A *sweep line* começa a varredura então, a partir do terceiro ponto do vetor.

Com a ordenação pelo ângulo, a *sweep line* faz um movimento anti-horário e quando passa por um ponto P_i ela analisa se ele mantém o sentido anti-horário com os dois elementos do topo da pilha. Se o sentido se mantém, o ponto é inserido em $C_H(S)$, caso contrário, o topo da pilha é removido e a verificação é feita novamente com P_i. Uma mudança de sentido é identificada quando o determinante matricial entre P_i e os dois pontos do topo da pilha é negativo. Ao final, o $C_H(S)$ terá todos os pontos em sentido anti-horário.

A solução está representada em pseudo-código no Algoritmo 3. Primeiro, é verificado se conjunto de pontos tem tamanho menor ou igual a 3, caso tenha, o envoltório é o próprio conjunto. Em seguida, acontece a escolha do pivô e a ordenação. Depois, os pontos são avaliados se fazem parte do envoltório convexo.

A implementação em C++ do Algoritmo 3 está apresentada na Seção C.1 e a

Algorithm 3 Envoltório Convexo de Graham.

Entradas: P = lista de *pontos*.
Saída: ch = lista de *pontos* que formam o envoltório convexo.

1: $N \leftarrow P$.size()
2:
3: "Ordene os pontos P por coordenada y crescente e, em caso de empate, por coordenada x decrescente"
4: $P \leftarrow sort_by_y_and_x(P)$
5:
6: **if** $N \leq 3$ **then**
7:     **return** P
8: **end if**
9:
10: "Coloque o pivô em $P[0]$"
11: $P[0] \leftarrow get_pivot(P)$
12:
13: "Ordene os pontos P pelo ângulo que formam com o pivô"
14: $P \leftarrow sort_by_angle(P)$
15:
16: ch.push($P[N-1]$)
17: ch.push($P[0]$)
18: ch.push($P[1]$)
19:
20: $i \leftarrow 2$
21: **while** $i < N$ **do**
22:     $j \leftarrow chi$.size() -1
23:     **if** get_determinant($ch[j-1], ch[j], P[i]$) > 0 **then**
24:         ch.push($P[i++]$)
25:     **else**
26:         ch.pop()
27:     **end if**
28: **end while**
29:
30: **return** ch

implementação em Python está apresentada em C.2. Em ambas, a ordenação foi feita usando as respectivas funções `sort()` de cada linguagem, mas passando para elas a regra de ordenação, que está definida na função `sort_by_angle()`. Nessa última função, para obter o ângulo entre os pontos e o pivô, foi utilizada a função `atan2()` das bibliotecas `math` de cada linguagem.

A Figura 13 mostra a *sweep line*, em azul, passando pelos pontos já ordenados. A linha preta representa o envoltório convexo. Na Figura 13a mostra quais são os três primeiros elementos do envoltório: o anterior ao pivô, o pivô, e o sucessor do pivô. Na Figura 13b, a *sweep line* passa pelo ponto 2 e verifica que ele, junto com o primeiro ponto e o pivô estão no mesmo sentido, nesse caso, ele é adicionado na pilha. Em seguida, na Figura 13c, agora é verificado que juntos, os pontos 1, 2 e 3 mantêm o sentido anti-horário, então o topo da pilha é atualizado com o ponto 3. Agora, na Figura 13d, os pontos analisados são os 2, 3, 4, nesse caso, o sentido entre eles não se manteve, então o topo da pilha, o ponto 3, é removido. Na Figura 13e, a *sweep line* avalia os pontos 4, 2, 1 e como eles mantêm o sentido, o ponto 4 é adicionado na pilha.

Essa verificação dos três pontos do topo da pilha continua na Figura 13f e na Figura 13g, até chegar no último ponto, o 6, que já estava no envoltório. A Figura 13h mostra o envoltório convexo encontrado.

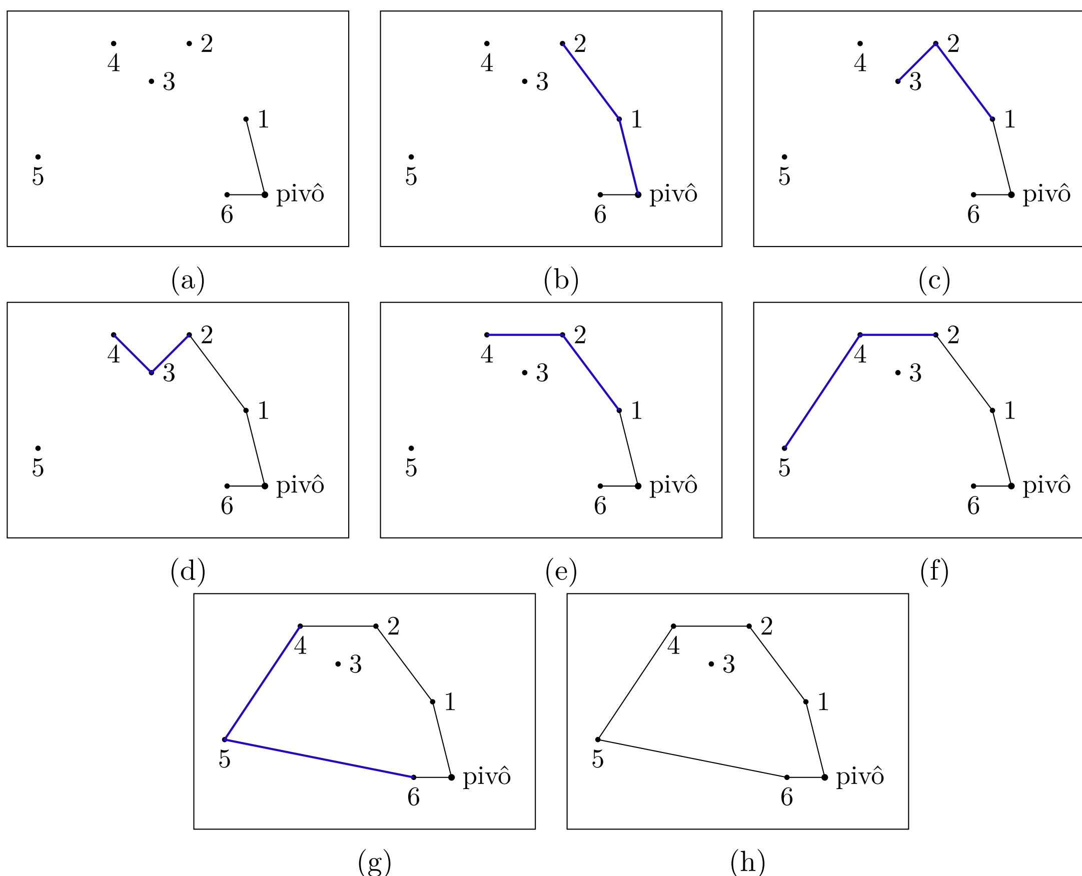


Figura 13 – *Sweep Line* no envoltório convexo de Graham.

3.3.2 Análise da complexidade

Para ordenar os pontos, a complexidade é de $O(N \log N)$. No *loop* principal, em que é calculado o envoltório convexo, para cada ponto, o algoritmo volta para verificar o sentido dele com os pontos anteriores, talvez isso faz parecer que esse *loop* tenha complexidade $O(N^2)$, mas cada ponto só é inserido ou removido uma única vez, então a complexidade é $O(N)$.

Como a complexidade da ordenação sobrepõe a complexidade de execução da *sweep line*, a complexidade geral do algoritmo é de $O(N \log N)$.

3.3.3 Análise da simplicidade de implementação

Os tamanhos das implementações em Python e C++ estão demonstrados na Tabela 4 com as métricas descritas na Seção 1.5. Para a contagem, só estão sendo consideradas as linhas que fazem parte da implementação do algoritmo: linhas 12 a 79, na implementação em C++; linhas 9 a 79, na implementação em Python.

As quantidades de linhas físicas e de linhas de código, em ambas as implementações, indicam que o algoritmo é grande para ser implementado. A quantidade de linhas efetivas de código foi maior na implementação em Python, mesmo que essa linguagem costume ser menos verbosa que C++. Alguns fatos podem ter contribuído pra isso, como, por exemplo, o fato de que na ordenação em C++ foi utilizada uma função anônima, mas em Python não é possível escrever funções anônimas em mais de uma linha, sendo necessário escrever uma função normal para manter a legibilidade do código.

As quantidades altas de linhas lógicas de código, representando a quantidade de instruções realizadas, mostra que o algoritmo não é de fácil memorização.

Para implementar esse algoritmo é preciso que o competidor tenha conhecimentos de geometria e álgebra linear, para saber como obter os ângulos, saber identificar pontos colineares e mudança de sentido, entre outros. Assim, a implementação do algoritmo não é simples e nem rápida.

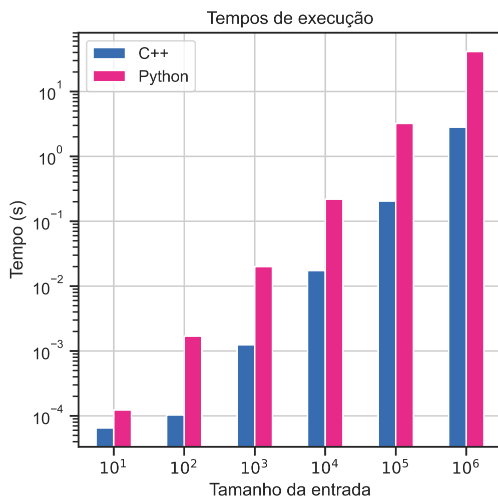
Tabela 4 – Métricas de tamanho de código para o algoritmo Graham.

Métrica	C++	Python
linhas de código	49	46
linhas físicas de código	68	71
linhas lógicas de código	27	30
linhas efetivas de código	40	46

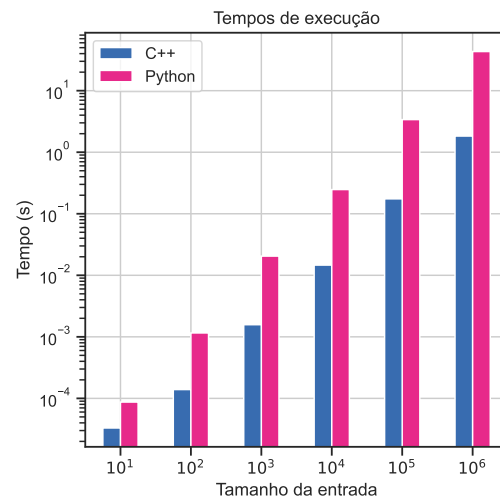
3.3.4 Comparação de tempo entre as implementações

A Figura 14 mostra as comparações de tempos, em escala logarítmica, das execuções em C++ e em Python para cada tipo de entrada de envoltório convexo descritos na Seção 2.4. Como esperado, para todos os casos e tamanhos de entrada, a execução em Python levou mais tempo. As maiores diferenças de tempos entre as implementações foram para as entradas com 10^6 pontos, em que os tempos em Python foram superiores aos tempos em C++, cerca de 14 vezes maior para o triângulo (Figura 14a) e para o polígono aleatório (Figura 14c) e cerca de 23 vezes maior para o retângulo (Figura 14b).

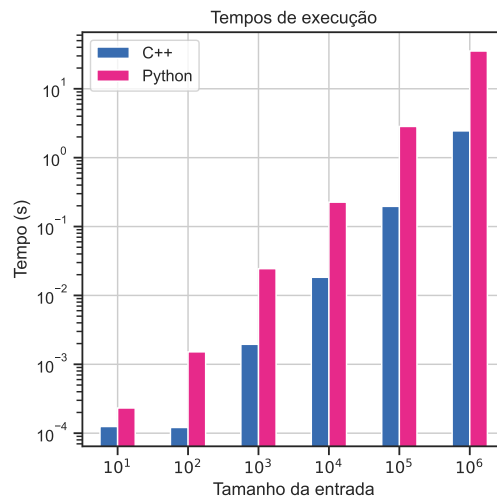
Para o contexto de competições de programação, é inviável usar a implementação em Python, proposta por este trabalho, para problemas com entradas de tamanho 10^6 ou superiores.



(a) Tempos para triângulo



(b) Tempos para retângulo



(c) Tempos para polígono aleatório

Figura 14 – Tempos de execução - Envoltório convexo de Graham.

3.3.5 Exemplo de problema

O problema *10652 - Board Wrapping* do Online Judge [3] pode ser resolvido com uma variação do algoritmo de Graham. Para encontrar os limites do polígono correspondentes ao envoltório convexo, depois de determinar a área total ocupada pelas placas, determinar os vértices de cada placa, assumindo que eles estão inicialmente com o centro na origem, então fazer a rotação em sentido horário e, em seguida, transladar os pontos para a posição correta.

3.4 Envoltório Convexo de Andrew

O algoritmo do envoltório convexo de Andrew, também conhecido como cadeia monótona, foi proposto em 1979 (ANDREW, 1979), e é uma alternativa ao algoritmo de Graham. O algoritmo determina os pontos do envoltório convexo $C_H(S)$, conforme Seção seção 1.4, com uma complexidade $O(N \log N)$. Ele constrói o envoltório em duas partes: envoltório superior e envoltório inferior.

3.4.1 Solução do problema com *sweep line*

Dado um conjunto finito S de N pontos, o problema é determinar o envoltório convexo $C_H(S)$. A ordenação é feita por coordenada x crescente e, em caso de empate, por coordenada y. Assim, a *sweep line* começa a varredura pelo ponto mais à direita com menor coordenada y. O $C_H(S)$ é gerado de forma semelhante ao algoritmo de Graham. O envoltório inferior é gerado empilhando os pontos em sequência, desde que o ponto analisado e os dois últimos pontos da pilha mantenham a orientação horária. O último ponto do envoltório inferior é o ponto mais à direita, com maior coordenada y, o qual também é o primeiro ponto do envoltório superior. Para gerar o envoltório superior é feito o mesmo procedimento, mas agora os pontos são processados em ordem decrescente e seguindo o sentido anti-horário. Ao final, os dois envoltórios devem ser unidos para formar o $C_H(S)$.

A solução está representada em pseudo-código no Algoritmo 4. São usados dois vetores, um para o envoltório superior e outro para o inferior. Depois dos pontos serem ordenados o envoltório inferior começa a ser formado, seguindo a mesma regra que o algoritmo de Graham: se o ponto mantém o sentido com os outros dois do topo da pilha, ele faz parte do envoltório. Para formar o envoltório superior, o conjunto de pontos ordenados é revertido, para seguir a ordem decrescente. O último ponto do envoltório inferior é removido pois ele é igual ao primeiro ponto do superior. Ao final, o $C_H(S)$ é composto pelos pontos dos dois envoltórios formados.

[3] <https://cutt.ly/wRPMegq>

Algorithm 4 Envoltório Convexo de Andrew.

Entradas: P = lista de *pontos*.
Saída: ch = lista de *pontos* que formam o envoltório convexo.

```
1: N ← P.size()
2: upper ← []
3: lower ← []
4:
5: "Ordene os pontos P por coordenadas x e y"
6: P ← sort_by_x_and_y(P)
7:
8: for p : P do
9:     size ← lower.size() −1
10:    while size ≥ 2 and get_determinant(lower[j − 1], lower[j], p) ≤ 0 do
11:        lower.pop()
12:        size ← lower.size() −1
13:    end while
14:
15:    lower.push(p)
16: end for
17:
18: reverse(P)
19:
20: for p : P do
21:     size ← upper.size() −1
22:     while size ≥ 2 and get_determinant(upper[j − 1], upper[j], p) ≤ 0 do
23:         upper.pop()
24:         size ← upper.size() −1
25:     end while
26:
27:     lower.push(p)
28: end for
29:
30: lower.pop()
31: ch = lower + upper
32:
33: return ch
```

A implementação em C++ do Algoritmo 4 está apresentada em D.1 e a implementação em Python está apresentada em D.2. Em ambas, a ordenação foi feita usando as respectivas funções `sort()` de cada linguagem, mas em C++ foi preciso definir o operador `<` na estrutura do ponto para que a coordenada x fosse priorizada.

Considerando os pontos da Figura 15, o envoltório inferior, representado pela linha vermelha, começa com os pontos 1 e 2. A *sweep line*, em azul na Figura, começa analisando se o ponto 3 mantém o sentido horário com os pontos 1 e 2 (Figura 15b), nesse caso, o sentido não se manteve, então o topo da pilha, o ponto 2, é removido e o 3 é adicionado (Figura 15c). Em seguida, o ponto 4 é analisado junto com os dois pontos do topo da pilha, o 3 e 1 , como o sentido se manteve, o 4 é adicionado à pilha (Figura 15d). A *sweep line* segue a sequência de pontos até encontrar o último, o ponto 7, formando o envoltório inferior (Figura 15j). Depois passa em sentido anti-horário, formando o convexo superior, em preto na Figura 15k, da mesma forma que no algoritmo de Graham (Figura 13).

Vale notar que os pontos da Figura 15 são os mesmos da Figura 13, mas estão enumerados diferentes por terem sido ordenados com regras diferentes.

3.4.2 Análise da complexidade

Para ordenar os pontos, a complexidade é de $O(N \log N)$. Mesmo que o $C_H(S)$ seja formado em duas partes, cada ponto só é considerado no máximo duas vezes em cada parte, então o envoltório convexo é formado com complexidade $O(N)$.

Como a complexidade da ordenação sobrepõe a complexidade de execução da *sweep line*, a complexidade geral do algoritmo é de $O(N \log N)$.

3.4.3 Análise da simplicidade de implementação

Os tamanhos das implementações em Python e C++ estão demonstrados na Tabela 5 com as métricas descritas na Seção 1.5. Para a contagem, só estão sendo consideradas as linhas que fazem parte da implementação do algoritmo: linhas 12 a 60, na implementação em C++; linhas 6 a 42, na implementação em Python.

As quantidades de linhas físicas e de linhas de código, em ambas as implementações, indicam que o algoritmo não é tão grande para ser implementado como o de Graham. A quantidade de linhas efetivas de código foi maior na implementação em C++, sendo mais verbosa do que a implementação em Python.

As quantidades de linhas lógicas de código, mostram que o algoritmo faz menos operações que o de Graham, sendo, assim, mais simples.

Para implementar esse algoritmo é preciso que o competidor tenha conhecimentos das propriedades de determinante. Assim, a implementação do algoritmo não é complexa

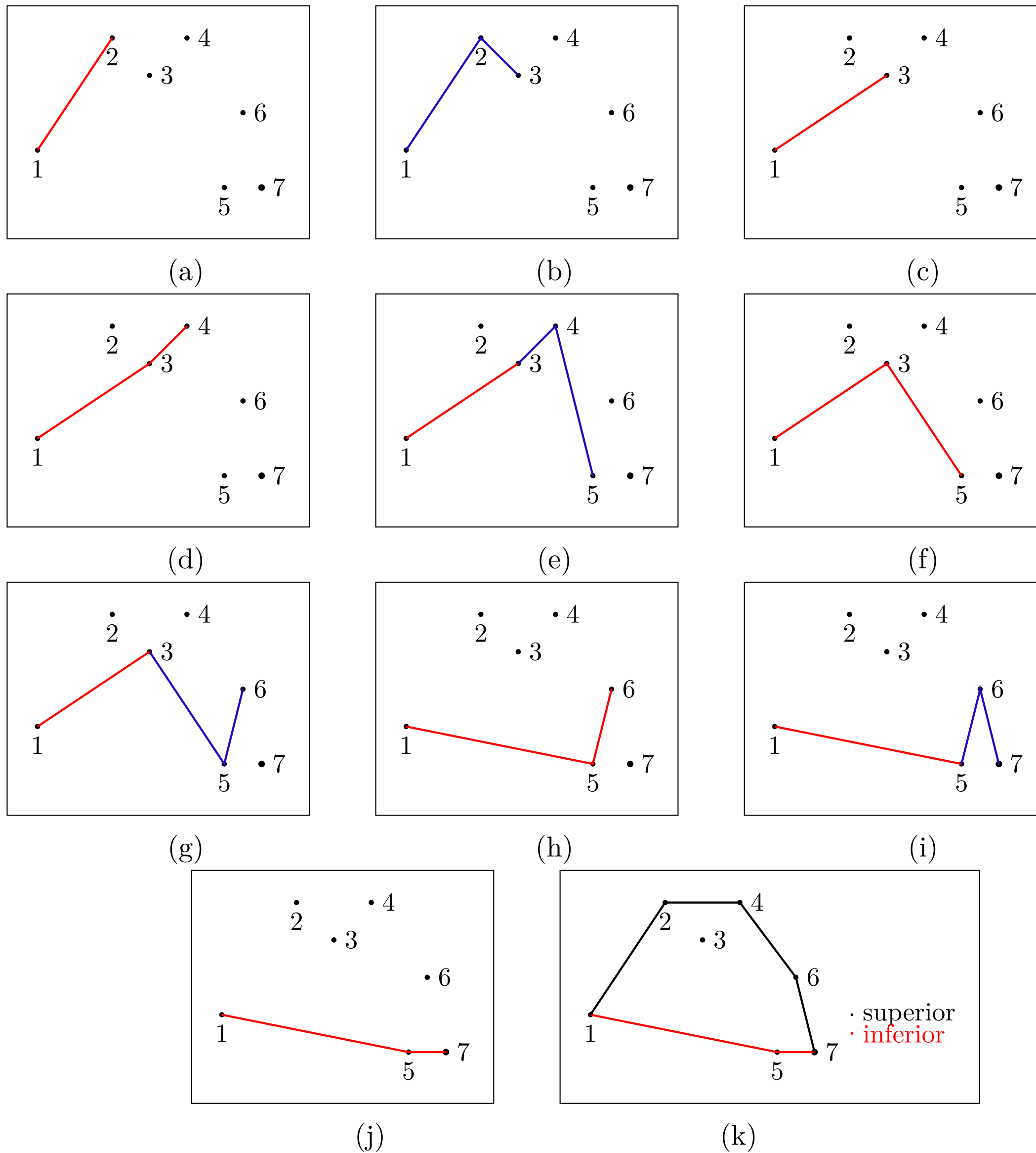


Figura 15 – *Sweep Line* no envoltório convexo de Andrew.

e nem demorada, comparada ao algoritmo de Graham.

Tabela 5 – Métricas de tamanho de código para o algoritmo Andrew.

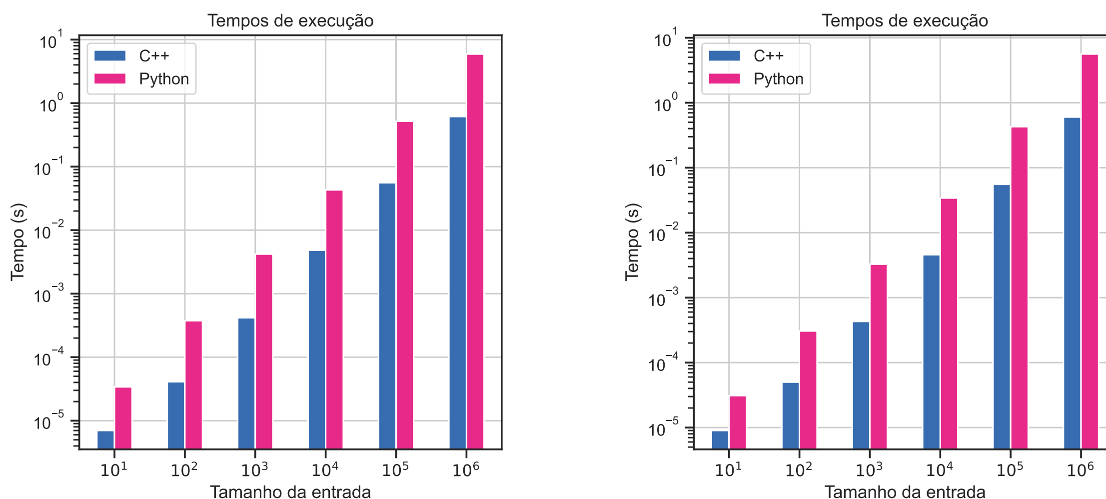
Métrica	**C++**	**Python**
linhas de código	34	23
linhas físicas de código	49	37
linhas lógicas de código	19	18
linhas efetivas de código	27	23

3.4.4 Comparação de tempo entre as implementações

A Figura 16 mostra as comparações de tempos, em escala logarítmica, das execuções em C++ e em Python para cada tipo de entrada de envoltório convexo descritos na Seção 2.4. Como esperado, para todos os casos e tamanhos de entrada, a execução

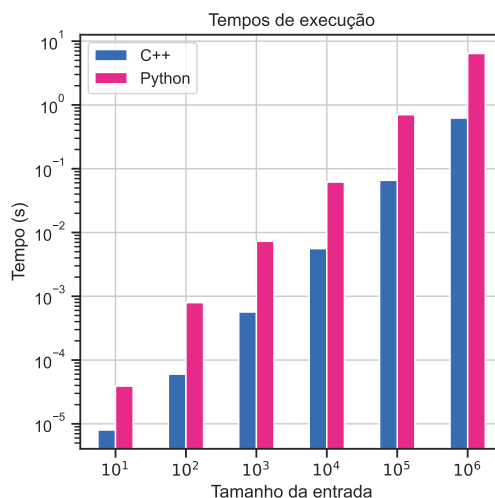
em Python levou mais tempo. Entretanto, em comparação ao algoritmo de Graham, os tempos foram bem menores.

Para o contexto de competições de programação, é viável utilizar as implementações propostas nas duas linguagens, visto que os tempos são razoáveis.



(a) Tempos para triângulo

(b) Tempos para retângulo



(c) Tempos para polígono aleatório

Figura 16 – Tempos de execução - Envoltório convexo de Andrew.

3.4.5 Exemplo de problema

O problema 1464 - Camadas de Cebola do Beecrowd[4] é um exemplo de como o algoritmo de Andrew pode ser usado. O problema consiste em encontrar o envoltório convexo do conjunto de pontos de entrada, excluir os pontos identificados, e reiniciar a rotina. A resposta dependerá da paridade de número de envoltórios computados.

[4] <https://www.urionlinejudge.com.br/judge/pt/problems/view/1464>

4 Considerações finais

Neste capítulo são apresentadas as considerações finais acerca do trabalho. Na Seção 4.1 são feitas algumas conclusões sobre os resultados do trabalho, já na Seção 4.2 algumas dificuldades encontradas são listadas. Por fim, na Seção 4.3 há propostas de trabalhos futuros.

4.1 Conclusões sobre os resultados

Os algoritmos usados em maratonas precisam ser de rápida implementação e eficientes, dado o contexto competitivo. Os algoritmos apresentados se mostraram eficientes para a maioria das entradas portanto satisfazem o requisito de eficiência, mas as implementações em Python propostas neste trabalho não são eficientes para as entradas com 10^6 elementos, então elas devem ser evitadas.

Algumas implementações não são simples e de fácil memorização, como é o caso do algoritmo de pontos de interseção de segmentos de retas e o do envoltório convexo de Graham, pois são grandes e requerem estruturas de dados especiais e conhecimentos não triviais de matemática. Seria ideal que os maratonistas levassem estes códigos impressos para que não percam tempo tentando lembrar dos detalhes da implementação. Assim, apenas os algoritmos de par de pontos mais próximos e o do envoltório convexo de Andrew atenderam aos dois requisitos.

4.2 Dificuldades encontradas

Em relação à execução do trabalho, a primeira dificuldade foi encontrar referências bibliográficas que explorem a *sweep line* como um objetivo ou como principal foco de trabalho.

Alguns fundamentos de geometria não estavam totalmente compreendidos no início do trabalho, o que afetou no tempo de compreensão dos algoritmos analisados e também na escrita. Outra dificuldade foi abstrair as ordenações e as complexidades dos algoritmos analisados, isso impactou na clareza ou falta de clareza na escrita.

No trabalho não foi investigado com que frequência aparecem problemas de geometria computacional em maratonas de programação para estimar melhor o quão útil é para os maratonistas conhecer o paradigma de *sweep line*, sendo assim, uma limitação do trabalho.

4.3 Trabalhos futuros

Como complemento ao trabalho, seria interessante investigar quais outros problemas podem ser solucionados com *sweep line* e também realizar implementações em outras linguagens para ter mais material de comparação.

Outro complemento interessante é fazer o levantamento da frequência com que problemas de geometria computacional aparecem em maratonas de programação, podendo fazer esse levantamento também para outros tópicos de problema.

Outro trabalho relevante seria analisar o uso de *sweep line* como solução para problemas e projetos reais, indo além do contexto de competições de programação.

Referências

ANDREW, A. M. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, v. 9, n. 5, p. 216–219, 1979. Citado na página 57.

ELMASRY, A.; KAMMER, F. Space-efficient plane-sweep algorithms. *CoRR*, abs/1507.01767, 2015. Disponível em: <http://arxiv.org/abs/1507.01767>. Citado 2 vezes nas páginas 35 e 47.

FENWICK, P. M. A new data structure for cumulative frequency tables. *SOFTWARE—PRACTICE AND EXPERIENCE*, v. 24, n. 3, p. 327–336, 1994. Citado na página 29.

GEEKSFORGEEKS. *IntroSort or Introspective sort*. 2021. <https://www.geeksforgeeks.org/introsort-or-introspective-sort/>. Acessado: 10-08-2021. Citado na página 47.

GEEKSFORGEEKS. *TimSort*. 2021. <https://www.geeksforgeeks.org/timsort/>. Acessado: 18-10-2021. Citado 2 vezes nas páginas 43 e 47.

GRAHAM, R. L. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, v. 1, n. 4, p. 132–133, 1972. Citado na página 51.

HALIM, S.; HALIM, F. *Competitive Programming 3*. 3th. ed. [S.l.]: Lulu.com, 2013. Citado na página 25.

ICPC. *Regional Rules*. 2020. <https://icpc.global/regionals/rules>. Acessado: 06-05-2021. Citado na página 25.

KEATING, K. Line segment intersection using a sweep line algorithm. Comp 163: Computational Geometry. 2005. Citado na página 27.

LAAKSONEN, A. *Competitive Programmer's Handbook*. 2018. Citado 4 vezes nas páginas 30, 35, 41 e 47.

SBC. *Sobre a Maratona de Programação*. 2020. <http://maratona.sbc.org.br/sobre20.html>. Acessado: 21-04-2021. Citado na página 25.

ZIVIANI, N. *Projeto de Algoritmos Com Implementações em Pascal e C*. 4th. ed. Brasil: Editora Pioneira, 1999. Citado na página 28.

APÊNDICE A – Códigos do algoritmo de pontos de interseção de segmentos de reta

A.1 C++

```
/*
Adaptação do código de Edson Alves, disponível em:
https://github.com/edsomjr/TEP/blob/master/Geometria_Computacional/slides/SL-1/points.cpp
*/

#include <bits/stdc++.h>

using namespace std;
using ll = long long int;

class BITree {
private:
    vector<ll> trees;
    size_t N;

    ll get_LSB(ll n){
        return n & (-n);
    }

public:
    BITree(size_t n) : trees(n + 1, 0), N(n) {}

    ll get_RSQ(ll i){
        ll sum = 0;

        while (i >= 1){
            sum += trees[i];
            i -= get_LSB(i);
        }

        return sum;
    }

    void add(size_t i, ll x){
        if (i == 0)
            return;

        while (i <= N){
            trees[i] += x;
```

```
40              i += get_LSB(i);
41          }
42      }
43  };
44
45  struct Point{
46      ll x, y;
47  };
48
49  struct Interval{
50      Point A, B;
51  };
52
53  ll get_index(const vector<ll>& hs, ll value){
54      auto it = lower_bound(hs.begin(), hs.end(), value);
55
56      return it - hs.begin() + 1;     // Contagem inicia em 1
57  }
58
59  ll count_intersections(const vector<Interval>& intervals){
60      struct Event {
61          ll type, x;
62          size_t idx;
63
64          bool operator<(const Event& e) const{
65              if (x != e.x)
66                  return x < e.x;
67
68              if (type != e.type)
69                  return type < e.type;
70
71              return idx < e.idx;
72          }
73      };
74
75      vector<Event> events;
76      set<ll> ys;        // Conjunto para compressão das coordenadas
77
78      for (size_t i = 0; i < intervals.size(); ++i){
79          auto I = intervals[i];
80
81          ys.insert(I.A.y);
82          ys.insert(I.B.y);
83
84          auto xmin = min(I.A.x, I.B.x);
85          auto xmax = max(I.A.x, I.B.x);
86
87          if (I.A.x == I.B.x)     // Vertical
88              events.push_back({2, xmin, i });
```

```
89          else{                   // Horizontal
90              events.push_back({1, xmin, i });
91              events.push_back({3, xmax, i });
92          }
93      }
94
95      sort(events.begin(), events.end());
96
97      vector<ll> hs(ys.begin(), ys.end());   // Mapa de compressão
98      BITree fenwick(hs.size());
99      ll total = 0;
100
101     for (const auto& event : events){
102         auto I = intervals[event.idx];
103
104         switch (event.type) {
105             case 1: {
106                 auto y = get_index(hs, I.A.y);
107                 fenwick.add(y, 1);
108             }
109                 break;
110
111             case 2: {
112                 auto ymin = min(get_index(hs, I.A.y), get_index(hs, I.B.y));
113                 auto ymax = max(get_index(hs, I.A.y), get_index(hs, I.B.y));
114                 total += fenwick.get_RSQ(ymax) - fenwick.get_RSQ(ymin - 1);
115             }
116                 break;
117
118             default: {
119                 auto y = get_index(hs, I.B.y);
120                 fenwick.add(y, -1);
121             }
122         }
123     }
124
125     return total;
126 }
127
128 int main(){
129     vector<Interval> intervals;
130
131     ll x_a, y_a;
132     ll x_b, y_b;
133
134     while(cin >> x_a >> y_a >> x_b >> y_b){
135         Point A = {x_a, y_a};
136         Point B = {x_b, y_b};
137
```

```
138            intervals.push_back({A, B});
139        }
140
141        auto begin = std::chrono::high_resolution_clock::now();
142
143        auto ans = count_intersections(intervals);
144
145        auto end = std::chrono::high_resolution_clock::now();
146        auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end - begin);
147
148        fprintf(stderr, "Time measured: %f seconds.\n", elapsed.count() * 1e-6);
149
150        cout << ans << '\n';
151
152        return 0;
153    }
```

Código 3 – Pontos de interseção de segmentos de reta - C++

A.2 Python

```
1   import sys
2   from bisect import bisect_left
3   from collections import namedtuple
4   import time
5
6
7   class BITree:
8       def __init__(self, size):
9           self.size = size
10          self.tree = [0] * (self.size + 1)
11
12      def get_LSB(self, n):
13          return n & -n
14
15      def get_RSQ(self, i):
16          sum_ = 0
17
18          while i >= 1:
19              sum_ += self.tree[i]
20              i -= self.get_LSB(i)
21
22          return sum_
23
24      def add(self, i, x):
25          if i == 0:
26              return
27
```

```
28          while i <= self.size:
29              self.tree[i] += x
30              i += self.get_LSB(i)
31
32
33  def get_index(heights, value):
34      index = bisect_left(heights, value)
35
36      return index + 1  # count start 1
37
38
39  def count_intersections(intervals):
40      events = []
41      heights = set()
42
43      for i in intervals:
44          heights.update([i.A.y, i.B.y])
45
46          x_min = min(i.A.x, i.B.x)
47          x_max = max(i.A.x, i.B.x)
48
49          index = intervals.index(i)
50
51          if i.A.x == i.B.x:  # vertical
52              events.append((x_min, 2, index))
53          else:  # horizontal
54              events.append((x_min, 1, index))
55              events.append((x_max, 3, index))
56
57      events.sort()
58      heights = sorted(heights)
59      fenwick_tree = BITree(len(heights))
60      total = 0
61
62      for e in events:
63          _, type_, idx = e
64          i = intervals[idx]
65
66          if type_ == 1:  # event type 1
67              y = get_index(heights, i.A.y)
68              fenwick_tree.add(y, 1)
69          elif type_ == 2:  # event type 2
70              y_min = min(get_index(heights, i.A.y), get_index(heights, i.B.y))
71              y_max = max(get_index(heights, i.A.y), get_index(heights, i.B.y))
72              total += fenwick_tree.get_RSQ(y_max) -\
73                  fenwick_tree.get_RSQ(y_min-1)
74          else:  # event type 3
75              y = get_index(heights, i.B.y)
76              fenwick_tree.add(y, -1)
```

```
77
78      return total
79
80
81  Point = namedtuple('Point', 'x y')
82  Interval = namedtuple('Interval', 'A B')
83
84  intervals = []
85
86  for line in sys.stdin:
87      x_a, y_a, x_b, y_b = map(int, line.split())
88      A = Point(x_a, y_a)
89      B = Point(x_b, y_b)
90      intervals.append(Interval(A, B))
91
92  begin = time.perf_counter()
93
94  answer = count_intersections(intervals)
95
96  end = time.perf_counter()
97  elapsed = (end - begin)
98
99  print(f"Time measured: {round(elapsed, 6)} seconds.", file=sys.stderr)
100
101 print(answer)
```

Código 4 – Pontos de interseção de segmentos de reta - Python

APÊNDICE B – Códigos do algoritmo de par de pontos mais próximos

B.1 C++

```
1  /*
2  Adaptação do código de Edson Alves, disponível em:
3  https://github.com/edsomjr/TEP/blob/master/Geometria_Computacional/slides/SL-2/closest.cpp
4  */
5
6  #include <bits/stdc++.h>
7
8  using namespace std;
9  using ll = long long int;
10 using point = pair<ll, ll>;
11
12 #define x first
13 #define y second
14
15 double get_distance(const point& P, const point& Q){
16     return hypot(P.x - Q.x, P.y - Q.y);
17 }
18
19 pair<point, point> get_closest_pair(vector<point>& ps){
20     size_t N = ps.size();
21     sort(ps.begin(), ps.end());
22
23     // Assume que N > 1
24     auto distance = get_distance(ps[0], ps[1]);
25     auto closest = make_pair(ps[0], ps[1]);
26
27     set<point> S;
28     S.insert(point(ps[0].y, ps[0].x));
29     S.insert(point(ps[1].y, ps[1].x));
30
31     for (ll i = 2; i < N; ++i){
32         auto P = ps[i];
33         auto it = S.lower_bound(point(P.y - distance, 0));
34
35         while (it != S.end()){
36             auto Q = point(it->second, it->first);
37
38             if (Q.x < P.x - distance){
39                 it = S.erase(it);
```

```
40                  continue;
41              }
42
43              if (Q.y > P.y + distance)
44                  break;
45
46              auto curr_distance = get_distance(P, Q);
47
48              if (curr_distance < distance){
49                  distance = curr_distance;
50                  closest = make_pair(P, Q);
51              }
52
53              ++it;
54          }
55
56          S.insert(point(P.y, P.x));
57      }
58
59      return closest;
60  }
61
62  int main(){
63      vector<point> P;
64      ll x, y = 0;
65
66      while(cin >> x >> y){
67          P.push_back({x, y});
68      }
69
70      auto begin = std::chrono::high_resolution_clock::now();
71
72      auto closest_pair = get_closest_pair(P);
73
74      auto end = std::chrono::high_resolution_clock::now();
75      auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end - begin);
76
77      fprintf(stderr, "Time measured: %f seconds.\n", elapsed.count() * 1e-6);
78
79      printf("%d %d\n%d %d\n",
80              closest_pair.first.first, closest_pair.first.second,
81              closest_pair.second.first, closest_pair.second.second
82      );
83
84      return 0;
85  }
```

Código 5 – Par de pontos mais próximos - C++

B.2 Python

```
1  import math
2  import sys
3  from collections import namedtuple
4  from bisect import bisect_left
5  import time
6
7
8  Point = namedtuple('Point', 'x y')
9  Pair = namedtuple('Pair', 'A B')
10
11
12 def get_distance(P, Q):
13     return math.hypot(P.x - Q.x, P.y - Q.y)
14
15
16 def get_closest_pair(points):
17     N = len(points)
18     points.sort()
19
20     distance = get_distance(points[0], points[1])
21     closest = Pair(points[0], points[1])
22
23     S = set()
24
25     S.add(Point(points[0].y, points[0].x))
26     S.add(Point(points[1].y, points[1].x))
27
28     S = sorted(S)
29
30     for i in range(2, N):
31         P = points[i]
32         index = bisect_left(S, Point(P.y - distance, 0))
33         size_s = len(S)
34
35         while index < size_s:
36             set_point = S[index]
37             Q = Point(set_point.y, set_point.x)
38
39             if Q.x < P.x - distance:
40                 S.remove(set_point)
41                 size_s = len(S)
42                 continue
43
44             if Q.y > P.y + distance:
45                 break
46
47             curr_distance = get_distance(P, Q)
```

```
48                 if curr_distance < distance:
49                     distance = curr_distance
50                     closest = Pair(P, Q)
51 
52                 index += 1
53 
54             S.append(Point(P.y, P.x))
55 
56     return closest
57 
58 
59 points = []
60 
61 for line in sys.stdin:
62     x, y = map(int, line.split())
63     points.append(Point(x, y))
64 
65 begin = time.perf_counter()
66 
67 closest_pair = get_closest_pair(points)
68 
69 end = time.perf_counter()
70 elapsed = (end - begin)
71 
72 print(f"Time measured: {round(elapsed, 6)} seconds.", file=sys.stderr)
73 
74 print(f"{closest_pair.A.x} {closest_pair.A.y}")
75 print(f"{closest_pair.B.x} {closest_pair.B.y}")
```

Código 6 – Par de pontos mais próximos - Python

APÊNDICE C – Códigos do algoritmo do envoltório convexo de Graham

C.1 C++

```
/*
Adaptação do código de Edson Alves, disponível em:
https://github.com/edsomjr/TEP/blob/master/Geometria_Computacional/slides/CH-1/graham.cpp
*/

#include <bits/stdc++.h>
#include <chrono>

using namespace std;
using ll = long long int;

template<typename T> struct Point{
    T x, y;

    double get_distance(const Point& P) const{
        return hypot(x - P.x, y - P.y);
    }
};

template<typename T> T get_determinant(const Point<T>& P, const Point<T>& A, const
↪   Point<T>& B){
    return (P.x * A.y + P.y * B.x + A.x * B.y) -
           (B.x * A.y + B.y * P.x + A.x * P.y);
}

template<typename T> Point<T> get_pivot(vector<Point<T>>& P){
    for (size_t i = 1; i < P.size(); ++i)
        if (P[i].y < P[0].y or
            (P[i].y == P[0].y and P[i].x > P[0].x))
                swap(P[0], P[i]);

    return P[0];
}

template<typename T> void sort_by_angle(vector<Point<T>>& P){
    auto P0 = get_pivot(P);

    sort(P.begin() + 1, P.end(),
        [&](const Point<T>& A, const Point<T>& B){
```

```
39              // pontos colineares: escolhe-se o mais próximo do pivô
40              if (get_determinant(P0, A, B) == 0) // se o get_determinante for 0, os
                ↪  pontos estão alinhados
41                  return A.get_distance(P0) < B.get_distance(P0);
42
43              auto alfa = atan2(A.y - P0.y, A.x - P0.x);
44              auto beta = atan2(B.y - P0.y, B.x - P0.x);
45
46              return alfa < beta;
47          }
48      );
49  }
50
51  template<typename T> vector<Point<T>> make_convex_hull(const vector<Point<T>>&
    ↪  points){
52      vector<Point<T>> P(points);
53      auto N = P.size();
54
55      // Corner case: com 3 vértices ou menos, P é o próprio convex hull
56      if (N <= 3)
57          return P;
58
59      sort_by_angle(P);
60
61      vector<Point<T>> ch;
62      ch.push_back(P[N - 1]); // o primeiro ponto é igual ao último
63      ch.push_back(P[0]);
64      ch.push_back(P[1]);
65
66      size_t i = 2;
67
68      while (i < N) {
69          auto j = ch.size() - 1;
70
71          // se o get_determinante for positivo, a orientação se manteve
72          if (get_determinant(ch[j - 1], ch[j], P[i]) > 0)
73              ch.push_back(P[i++]);
74          else
75              ch.pop_back();
76      }
77
78      return ch;
79  }
80
81  int main(){
82      vector<Point<ll>> P;
83      ll x, y = 0;
84
85      while(cin >> x >> y){
```

```
86          P.push_back({x, y});
87      }
88
89      auto begin = std::chrono::high_resolution_clock::now();
90
91      auto ch = make_convex_hull(P);
92
93      auto end = std::chrono::high_resolution_clock::now();
94      auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end - begin);
95
96      fprintf(stderr, "Time measured: %f seconds.\n", elapsed.count() * 1e-6);
97
98
99      for (size_t i = 0; i < ch.size(); ++i)
100         cout << ch[i].x << " " << ch[i].y << endl;
101
102     return 0;
103 }
```

Código 7 – Envoltório Convexo de Graham - C++

C.2 Python

```
1  #!/usr/bin/env python3
2  import math
3  import sys
4  from functools import cmp_to_key
5  from collections import namedtuple
6  import time
7
8
9  P0 = ()
10
11
12 def get_distance(get_pivot, point):
13     return math.hypot(point.x - get_pivot.x, point.y - get_pivot.y)
14
15
16 def get_determinant(P, A, B):
17     return (P.x * A.y + P.y * B.x + A.x * B.y) -\
18            (B.x * A.y + B.y * P.x + A.x * P.y)
19
20
21 def get_pivot(P):
22     i = 0
23     while i < len(P):
24         if P[i].y < P[0].y or (P[i].y == P[0].y and P[i].x > P[0].x):
25             P[i], P[0] = P[0], P[i]
```

```
26          i += 1
27      return P[0]
28
29
30  def sort_by_angle(A, B):
31      # pontos colineares: escolhe-se o mais próximo do pivô
32      # se o get_determinante for 0, os pontos estão alinhados
33      if get_determinant(P0, A, B) == 0:
34          if get_distance(P0, A) < get_distance(P0, B):
35              return -1
36          if get_distance(P0, A) > get_distance(P0, B):
37              return 1
38          return 0
39
40      alfa = math.atan2(A.y - P0.y, A.x - P0.x)
41      beta = math.atan2(B.y - P0.y, B.x - P0.x)
42
43      if alfa < beta:
44          return -1
45      if alfa > beta:
46          return 1
47      return 0
48
49
50  def make_convex_hull(P):
51      N = len(P)
52
53      # Corner case: com 3 vértices ou menos, P é o próprio convex hull
54      if N <= 3:
55          return P
56
57      global P0
58      P0 = get_pivot(P)
59      P.remove(P0)
60
61      P.sort(key=cmp_to_key(sort_by_angle))
62
63      N = len(P)
64
65      # o primeiro ponto é igual ao último
66      ch = [P[N-1], P0, P[0]]
67
68      i = 1
69      while i < N:
70          j = len(ch) - 1
71
72          # se o get_determinante for positivo, a orientação de manteve
73          if get_determinant(ch[j-1], ch[j], P[i]) > 0:
74              ch.append(P[i])
```

```
75              i += 1
76          else:
77              ch.pop()
78
79      return ch
80
81
82  Point = namedtuple('Point', 'x y')
83
84  P = []
85
86  for line in sys.stdin:
87      x, y = map(int, line.split())
88      P.append(Point(x, y))
89
90  begin = time.perf_counter()
91
92  ch = make_convex_hull(P)
93
94  end = time.perf_counter()
95  elapsed = (end - begin)
96
97  print(f"Time measured: {round(elapsed, 6)} seconds.", file=sys.stderr)
98
99  i = 0
100 for i in range(len(ch)):
101     print(f"{ch[i].x} {ch[i].y}")
```

Código 8 – Envoltório Convexo de Graham - Python

APÊNDICE D – Códigos do algoritmo do envoltório convexo de Andrew

D.1 C++

```
/*
Adaptação do código de Edson Alves, disponível em:
https://github.com/edsomjr/TEP/blob/master/Geometria_Computacional/slides/CH-1/andrew.cpp
*/

#include <bits/stdc++.h>
#include <chrono>

using namespace std;
using ll = long long int;

template<typename T> struct Point{
    T x, y;

    bool operator<(const Point& P) const{
        return x == P.x ? y < P.y : x < P.x;
    }
};

template<typename T> T get_determinant(const Point<T>& P, const Point<T>& A, const
↪   Point<T>& B){
    return (P.x * A.y + P.y * B.x + A.x * B.y) -
           (B.x * A.y + B.y * P.x + A.x * P.y);
}

template<typename T>
vector<Point<T>> make_hull(const vector<Point<T>>& points, vector<Point<T>>& hull){
    vector<Point<T>> P(points);

    for (const auto& p : P){
        auto size = hull.size();

        while (size >= 2 and get_determinant(hull[size - 2], hull[size - 1], p) <= 0){
            hull.pop_back();
            size = hull.size();
        }

        hull.push_back(p);
    }
```

```
39
40     return hull;
41 }
42
43 template<typename T> vector<Point<T>> make_monotone_chain(const vector<Point<T>>&
   ↪ points){
44     vector<Point<T>> P(points);
45
46     sort(P.begin(), P.end());
47
48     vector<Point<T>> lower_hull, upper_hull;
49
50     lower_hull = make_hull(P, lower_hull);
51
52     reverse(P.begin(), P.end());
53
54     upper_hull = make_hull(P, upper_hull);
55
56     lower_hull.pop_back();
57     lower_hull.insert(lower_hull.end(), upper_hull.begin(), upper_hull.end());
58
59     return lower_hull;
60 }
61
62 int main(){
63     vector<Point<ll>> P;
64     ll x, y = 0;
65
66
67     while(cin >> x >> y){
68         P.push_back({x, y});
69     }
70
71     auto begin = std::chrono::high_resolution_clock::now();
72
73     auto ch = make_monotone_chain<ll>(P);
74
75     auto end = std::chrono::high_resolution_clock::now();
76     auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end - begin);
77
78     fprintf(stderr, "Time measured: %f seconds.\n", elapsed.count() * 1e-6);
79
80     for (size_t i = 0; i < ch.size(); ++i)
81         cout << ch[i].x << " " << ch[i].y << endl;
82
83     return 0;
84 }
```

Código 9 – Envoltório Convexo de Andrew - C++

D.2 Python

```
from collections import namedtuple
import sys
import time


def get_determinant(P, A, B):
    return (P.x * A.y + P.y * B.x + A.x * B.y) -\
           (B.x * A.y + B.y * P.x + A.x * P.y)


def make_hull(P):
    hull = []

    for p in P:
        size = len(hull)

        while size >= 2 and get_determinant(
                hull[size-2], hull[size-1], p) <= 0:
            hull.pop()
            size = len(hull)

        hull.append(p)

    return hull


def make_monotone_chain(P):
    P.sort()

    lower_hull, upper_hull = [], []

    lower_hull = make_hull(P)

    P.reverse()

    upper_hull = make_hull(P)

    if lower_hull:
        lower_hull.pop()
    lower_hull.extend(upper_hull)

    return lower_hull


Point = namedtuple('Point', 'x y')

P = []
```

```
48
49 for line in sys.stdin:
50     x, y = map(int, line.split())
51     P.append(Point(x, y))
52
53 begin = time.perf_counter()
54
55 ch = make_monotone_chain(P)
56
57 end = time.perf_counter()
58 elapsed = (end - begin)
59
60 i = 0
61 for i in range(len(ch)):
62     print(f"{ch[i].x} {ch[i].y}")
```

Código 10 – Envoltório Convexo de Andrew - Python

APÊNDICE E – *Scripts* para gerar entradas do algoritmo de interseção de segmentos de reta

E.1 Melhor caso

```
import random
from sys import stderr
from collections import namedtuple

K = 100
Point = namedtuple('Point', 'x y')
Interval = namedtuple('Interval', 'A B')

print("Give the quantity of segments: ", file=stderr, end="")
size = int(input())


def gen_lines():
    intervals = []

    length = random.randint(K/2, K)

    v_y_a = random.randint(1, K)
    v_y_b = v_y_a + length

    h_x_a = random.randint(1, K)
    h_x_b = h_x_a + length

    h_y_a = random.randint(v_y_a, v_y_b)
    h_y_b = h_y_a

    A = Point(h_x_a, h_y_a)
    B = Point(h_x_b, h_y_b)

    intervals.append(Interval(A, B))  # horizontal

    for i in range(size-1):  # verticals
        x_a = random.randint(h_x_a, h_x_b)
        x_b = x_a

        A = Point(x_a, v_y_a)
```

```
37          B = Point(x_b, v_y_b)
38
39          intervals.append(Interval(A, B))
40
41          print(len(intervals), file=stderr)
42      return intervals
43
44
45  intervals = gen_lines()
46  for i in intervals:
47      print(f"{i.A.x} {i.A.y} {i.B.x} {i.B.y}")
```

Código 11 – Script para melhor caso do algoritmo de interseção de segmentos de reta

E.2 Pior caso

```
1   import random
2   from sys import stderr
3   from collections import namedtuple
4
5   K = 100
6   Point = namedtuple('Point', 'x y')
7   Interval = namedtuple('Interval', 'A B')
8
9   print("Give the quantity of segments: ", file=stderr, end="")
10  size = int(input())
11
12
13  def gen_lines():
14      intervals = set()
15
16      length = random.randint(size/2, max(K, size))
17      print(length, file=stderr)
18
19      h_x_a = random.randint(1, size)
20      h_x_b = h_x_a + length
21
22      v_y_a = random.randint(1, size)
23      v_y_b = v_y_a + length
24
25      while len(intervals) < size / 2:  # verticals
26          x_a = random.randint(h_x_a, h_x_b)
27          x_b = x_a
28
29          A = Point(x_a, v_y_a)
30          B = Point(x_b, v_y_b)
31
32          intervals.add(Interval(A, B))
```

```
33
34      print(len(intervals), file=stderr)
35
36      while len(intervals) < size:  # horizontals
37          y_a = random.randint(v_y_a, v_y_b)
38          y_b = y_a
39
40          A = Point(h_x_a, y_a)
41          B = Point(h_x_b, y_b)
42
43          intervals.add(Interval(A, B))
44
45      print(len(intervals), file=stderr)
46
47      return intervals
48
49
50  intervals = gen_lines()
51  for i in intervals:
52      print(f"{i.A.x} {i.A.y} {i.B.x} {i.B.y}")
```

Código 12 – Script para pior caso do algoritmo de interseção de segmentos de reta

APÊNDICE F – *Scripts* para gerar entradas do algoritmo de par de pontos mais próximos

F.1 Melhor caso

```
1  import random
2  from sys import stderr
3
4  print("Give the size of the set of points: ", file=stderr, end="")
5  size = int(input())
6
7  K = 100
8
9
10 def gen_points():
11     S = set()
12
13     y = 1
14
15     while len(S) < size:
16         x = random.randint(-K, K*size)
17         S.add((x, y))
18         print(len(S), file=stderr)
19
20     return S
21
22
23 S = gen_points()
24 for p in S:
25     print(f"{p[0]} {p[1]}")
```

Código 13 – Script para melhor caso do algoritmo de par de pontos mais próximos

F.2 Pior caso

```
1  import random
2  from sys import stderr
3
4  K = 100
5
6  print("Give the size of the set of points: ", file=stderr, end="")
7  size = int(input())
```

```
8   size_lines = size/2
9
10
11  def gen_points():
12      S = set()
13
14      x_1 = 1
15      y_1 = 1
16
17      x_2 = random.randint(x_1+2, K)
18
19      S.add((x_1, y_1))
20      S.add((x_2, y_1))
21
22      while len(S) < size:
23          y = random.randint(y_1, K*size)
24          S.add((x_1, y))
25          S.add((x_2, y))
26
27          print(len(S), file=stderr)
28
29      return S
30
31
32  S = gen_points()
33  for p in S:
34      print(f"{p[0]} {p[1]}")
```

Código 14 – Script para pior caso do algoritmo de par de pontos mais próximos

APÊNDICE G – *Scripts* para gerar entradas dos algoritmos de envoltório convexo

G.1 Triângulo

```
import random
import math
import matplotlib.path as mplPath
import numpy as np
from sys import stderr

Ax, Ay = 0, 0
Bx, By = 0, 0
Cx, Cy = 0, 0

K = 100

print("Give the size of the set of points: ", file=stderr, end="")
size = int(input())


def is_triangle(a, b, c):
    return a + b > c and a + c > b and b + c > a


def gen_triangle():
    global Ax, Ay, Bx, By, Cx, Cy

    Ax = random.randint(-K*size, K*size)
    Ay = random.randint(-K*size, K*size)
    Bx = random.randint(-K*size, K*size)
    By = random.randint(-K*size, K*size)
    Cx = random.randint(-K*size, K*size)
    Cy = random.randint(-K*size, K*size)

    a = math.hypot(Ax-Bx, Ay-By)
    b = math.hypot(Bx-Cx, By-Cy)
    c = math.hypot(Cx-Ax, Cy-Ay)

    return a, b, c


def calc_area(a, b, c):
    s = (a + b + c)/2
```

```
40      area = math.sqrt(s*(s-a)*(s-b)*(s-c))  # Heron's formula
41      # print("area:", area)
42      return area
43
44
45  def gen_points():
46      S = set()
47      S.add((Ax, Ay))
48      S.add((Bx, By))
49      S.add((Cx, Cy))
50
51      polygon = mplPath.Path(
52          np.array([[Ax, Ay], [Bx, By], [Cx, Cy]])
53      )
54
55      while True:
56          Px = random.randint(-K*size, K*size)
57          Py = random.randint(-K*size, K*size)
58
59          if polygon.contains_point((Px, Py)):
60              S.add((Px, Py))
61              print(len(S), file=stderr)
62
63          if len(S) >= size:
64              break
65
66      return S
67
68
69  a, b, c = gen_triangle()
70  while not is_triangle(a, b, c) or calc_area(a, b, c) < K*K*size:
71      a, b, c = gen_triangle()
72
73  # print("sides", a, b, c)
74  # print("Ponto A:", Ax, Ay)
75  # print("Ponto B:", Bx, By)
76  # print("Ponto C:", Cx, Cy)
77
78  S = gen_points()
79  # print("conjunto S")
80  for p in S:
81      print(f"{p[0]} {p[1]}")
```

Código 15 – Script para triângulo como envoltório convexo

G.2 Retângulo

```
import random
from sys import stderr

Ax, Ay = 0, 0
Bx, By = 0, 0
Cx, Cy = 0, 0
Dx, Dy = 0, 0

K = 1000

print("Give the size of the set of points: ", file=stderr, end="")
size = int(input())


def gen_rectangle():
    global Ax, Ay, Bx, By, Cx, Cy, Dx, Dy

    Ax = random.randint(-K*size, K*size)
    Ay = random.randint(-K*size, K*size)

    h = random.randint(1, K*size)
    w = random.randint(1, K*size)

    Bx = Ax + w
    By = Ay

    Cx = Bx
    Cy = By - h

    Dx = Cx - w
    Dy = Cy

    return h, w


def calc_area(h, w):
    return h * w


def gen_points():
    S = set()
    S.add((Ax, Ay))
    S.add((Bx, By))
    S.add((Cx, Cy))
    S.add((Dx, Dy))

    while len(S) < size:
```

```
48          Px = random.randint(Ax, Bx)
49          Py = Ay
50          S.add((Px, Py))  # side A-B
51
52          if len(S) < size:
53              Px = Bx
54              Py = random.randint(Cy, By)
55              S.add((Px, Py))  # side C-B
56
57          if len(S) < size:
58              Px = random.randint(Dx, Cx)
59              Py = Dy
60              S.add((Px, Py))  # side D-C
61
62          if len(S) < size:
63              Px = Ax
64              Py = random.randint(Dy, Ay)
65              S.add((Px, Py))  # side D-A
66
67          print(len(S), file=stderr)
68
69      return S
70
71
72  h, w = gen_rectangle()
73  ratio = min(h, w)/max(h, w)
74  while calc_area(h, w) < K*K*size or ratio < 0.25:
75      h, w = gen_rectangle()
76      ratio = min(h, w)/max(h, w)
77
78  S = gen_points()
79  for p in S:
80      print(f"{p[0]} {p[1]}")
```

Código 16 – Script para retângulo como envoltório convexo

G.3 Polígono aleatório

```
1   import random
2   from sys import stderr
3
4   Ax, Ay = 0, 0
5
6   K = 100
7
8   print("Give the size of the set of points: ", file=stderr, end="")
9   size = int(input())
10
```

```
11
12 def gen_points():
13     S = set()
14     S.add((Ax, Ay))
15
16     while True:
17         Px = random.randint(-K*size, K*size)
18         Py = random.randint(-K*size, K*size)
19         S.add((Px, Py))
20
21         print(len(S), file=stderr)
22
23         if len(S) >= size:
24             break
25
26     return S
27
28
29 S = gen_points()
30 for p in S:
31     print(f"{p[0]} {p[1]}")
```

Código 17 – Script para polígono aleatório como envoltório convexo