Geometria Computacional

Sweep line: algoritmos

Prof. Edson Alves

Faculdade UnB Gama

Sumário

Par de pontos mais próximo

Par de pontos mais próximo

- Dado um conjunto S de N de pontos no plano bidimensional, o problema de encontrar o par de pontos mais próximo consiste em encontrar dois pontos $P, Q \in S$ tal que

$$\text{dist}(P, Q) = \min\{\text{dist}(P_i, P_j)\}, \quad \forall P_i \in S \quad \text{com} \quad i \neq j$$

Par de pontos mais próximo

- Dado um conjunto S de N de pontos no plano bidimensional, o problema de encontrar o par de pontos mais próximo consiste em encontrar dois pontos $P, Q \in S$ tal que

$$\text{dist}(P, Q) = \min\{\text{dist}(P_i, P_j)\}, \quad \forall P_i \in S \quad \text{com} \quad i \neq j$$

- Uma abordagem de busca completa consiste em computar as distância entre todos os pares de pontos possível, tendo complexidade $O(N^2)$

Par de pontos mais próximo

- Dado um conjunto S de N de pontos no plano bidimensional, o problema de encontrar o par de pontos mais próximo consiste em encontrar dois pontos $P, Q \in S$ tal que

$$\text{dist}(P, Q) = \min\{\text{dist}(P_i, P_j)\}, \quad \forall P_i \in S \quad \text{com} \quad i \neq j$$

- Uma abordagem de busca completa consiste em computar as distância entre todos os pares de pontos possível, tendo complexidade $O(N^2)$
- Contudo, o problema pode ser resolvido em $O(N \log N)$ através do *sweep line*

Par de pontos mais próximo

- Dado um conjunto S de N de pontos no plano bidimensional, o problema de encontrar o par de pontos mais próximo consiste em encontrar dois pontos $P, Q \in S$ tal que

$$\text{dist}(P, Q) = \min\{\text{dist}(P_i, P_j)\}, \quad \forall P_i \in S \quad \text{com} \quad i \neq j$$

- Uma abordagem de busca completa consiste em computar as distância entre todos os pares de pontos possível, tendo complexidade $O(N^2)$
- Contudo, o problema pode ser resolvido em $O(N \log N)$ através do *sweep line*
- Os pontos deve ser ordenados em ordem lexicográfica

Par de pontos mais próximo

- Seja $d = \text{dist}(P_1, P_2)$

Par de pontos mais próximo

- Seja $d = \text{dist}(P_1, P_2)$
- Agora, para todos pontos $P_3, P_4, \ldots, P_N$, deve-se computar todos os pontos vizinhos de $P_i = (x, y)$ tais que as coordenadas x estejam no intervalo $[x - d, x]$ e que as coordenadas y estejam no intervalo $[y - d, y + d]$

Par de pontos mais próximo

- Seja $d = \text{dist}(P_1, P_2)$
- Agora, para todos pontos $P_3, P_4, \ldots, P_N$, deve-se computar todos os pontos vizinhos de $P_i = (x, y)$ tais que as coordenadas x estejam no intervalo $[x - d, x]$ e que as coordenadas y estejam no intervalo $[y - d, y + d]$
- Estes pontos podem ser identificados mantendo-se um conjunto de pontos cujas coordenadas estejam entre $[x - d, x]$, ordenado em ordem crescente de coordenada y

Par de pontos mais próximo

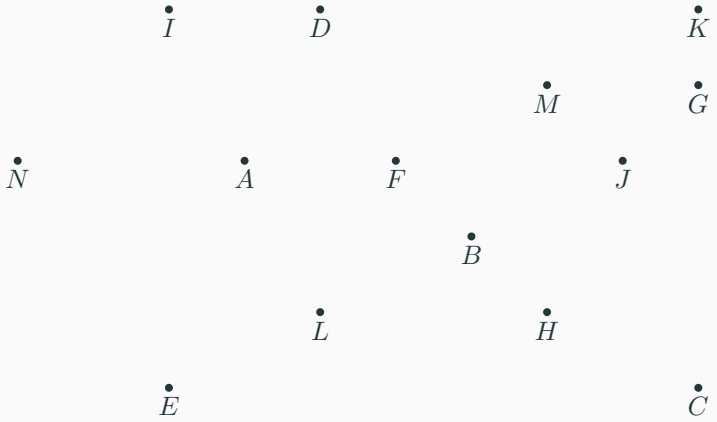
- Seja $d = \text{dist}(P_1, P_2)$
- Agora, para todos pontos $P_3, P_4, \ldots, P_N$, deve-se computar todos os pontos vizinhos de $P_i = (x, y)$ tais que as coordenadas x estejam no intervalo $[x - d, x]$ e que as coordenadas y estejam no intervalo $[y - d, y + d]$
- Estes pontos podem ser identificados mantendo-se um conjunto de pontos cujas coordenadas estejam entre $[x - d, x]$, ordenado em ordem crescente de coordenada 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

Par de pontos mais próximo

- Seja $d = \text{dist}(P_1, P_2)$
- Agora, para todos pontos $P_3, P_4, \ldots, P_N$, deve-se computar todos os pontos vizinhos de $P_i = (x, y)$ tais que as coordenadas x estejam no intervalo $[x - d, x]$ e que as coordenadas y estejam no intervalo $[y - d, y + d]$
- Estes pontos podem ser identificados mantendo-se um conjunto de pontos cujas coordenadas estejam entre $[x - d, x]$, ordenado em ordem crescente de coordenada 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
- O ponto principal é que existem, no máximo, $O(1)$ pontos neste retângulo, o que faz com que a complexidade do algoritmo seja $O(N \log N)$, por conta da ordenação

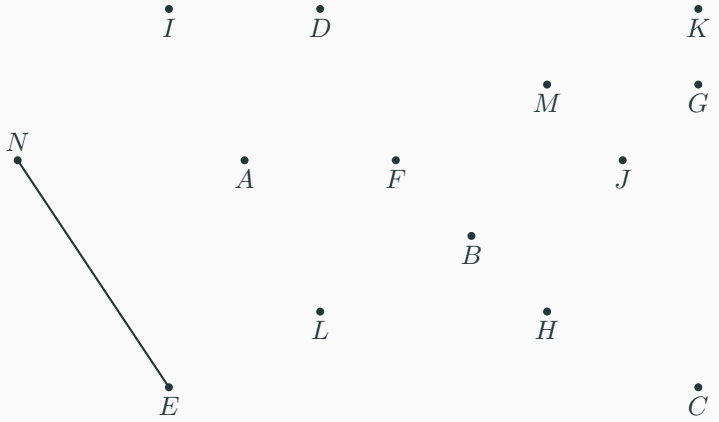
Visualização de identificação do par de pontos mais próximo

Par mais próximo: –



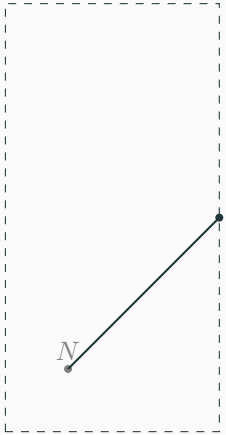
Visualização de identificação do par de pontos mais próximo

Par inicial, $\mathrm{dist}(N, E) = \mathbf{3.605551}$

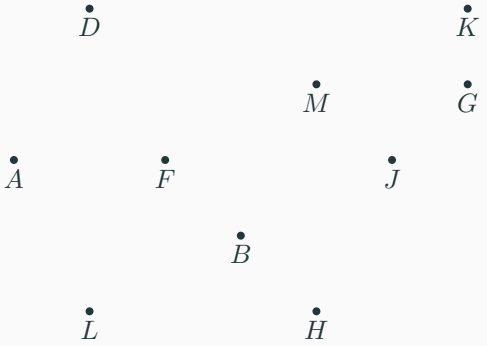


Visualização de identificação do par de pontos mais próximo

Visualização de identificação do par de pontos mais próximo



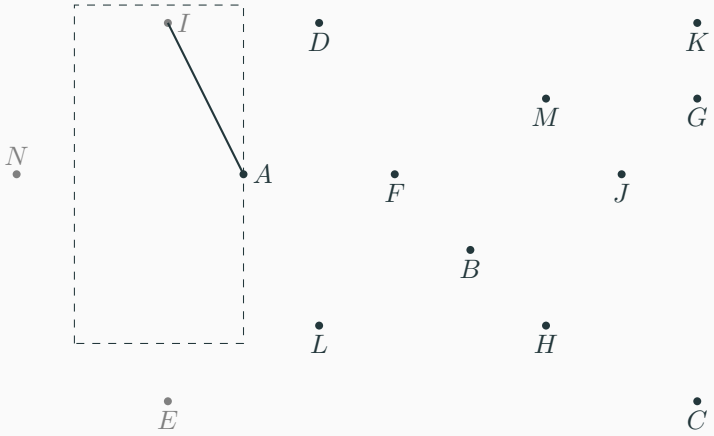
$$\text{dist}(I, N) = \mathbf{2.828427} < \text{dist}(N, E) = 3.605551$$



Visualização de identificação do par de pontos mais próximo

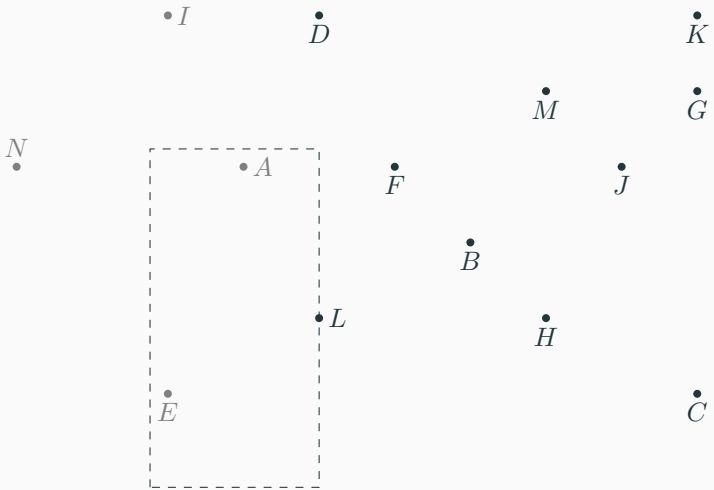
Visualização de identificação do par de pontos mais próximo

$$\mathrm{dist}(A, I) = \mathbf{2.236068} < \mathrm{dist}(I, N) = 2.828427$$



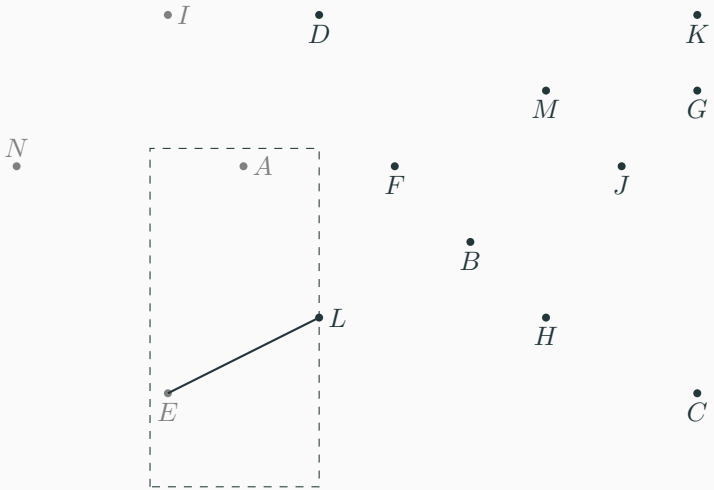
Visualização de identificação do par de pontos mais próximo

Avaliação do ponto L



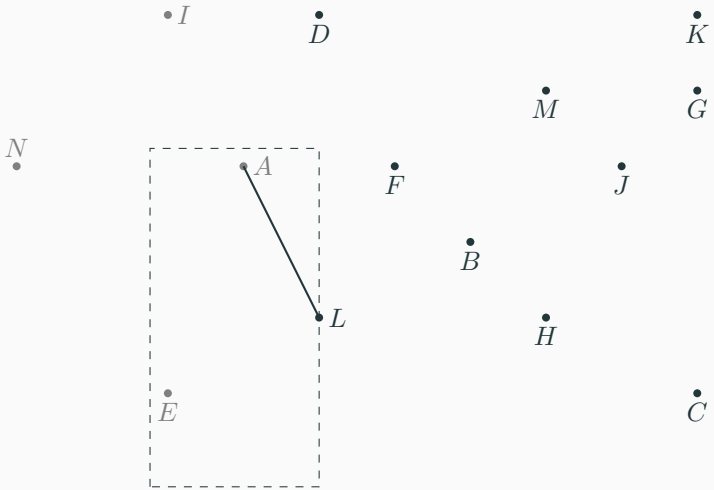
Visualização de identificação do par de pontos mais próximo

$$\mathrm{dist}(L, E) = \mathrm{dist}(A, I) = \mathbf{2.236068}$$



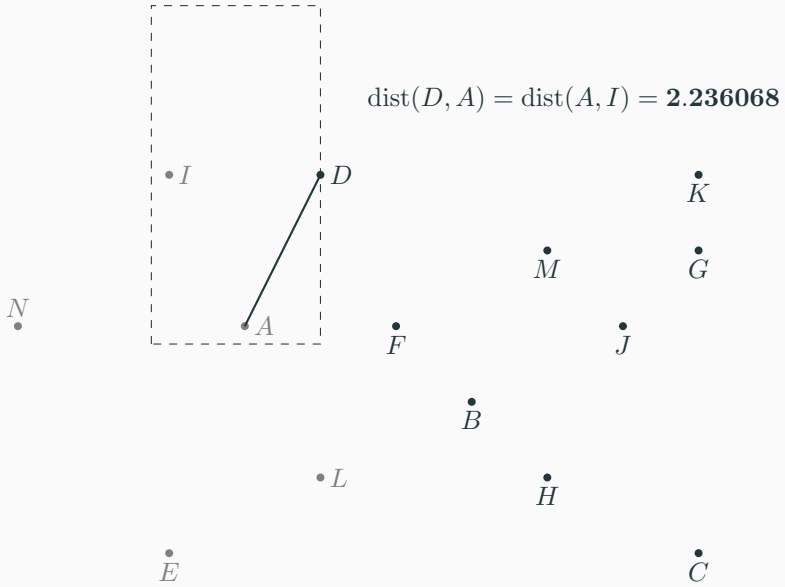
Visualização de identificação do par de pontos mais próximo

$$\mathrm{dist}(L, A) = \mathrm{dist}(A, I) = \mathbf{2.236068}$$

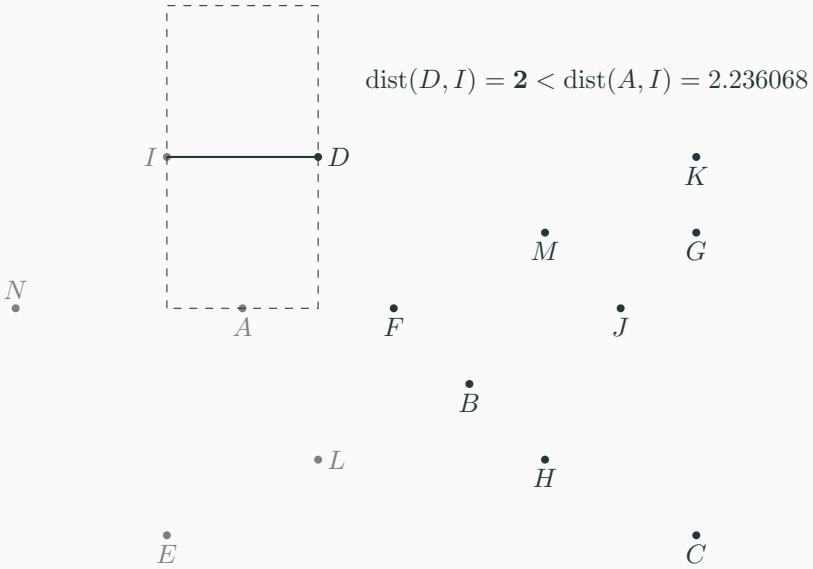


Visualização de identificação do par de pontos mais próximo

Visualização de identificação do par de pontos mais próximo



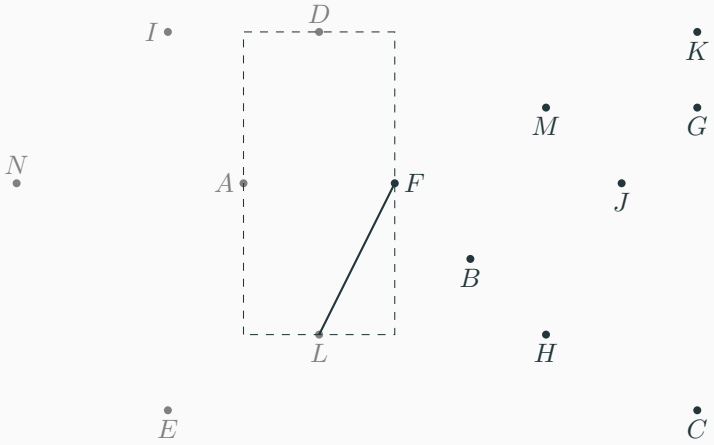
Visualização de identificação do par de pontos mais próximo



Visualização de identificação do par de pontos mais próximo

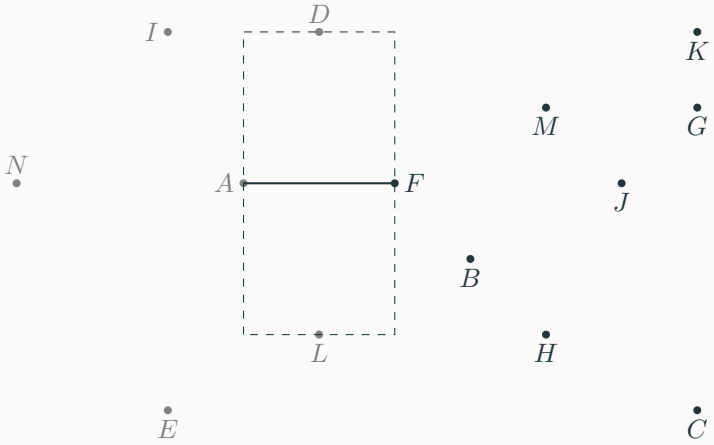
Visualização de identificação do par de pontos mais próximo

$$\text{dist}(F, L) = 2.236068 > \text{dist}(D, I) = \mathbf{2}$$



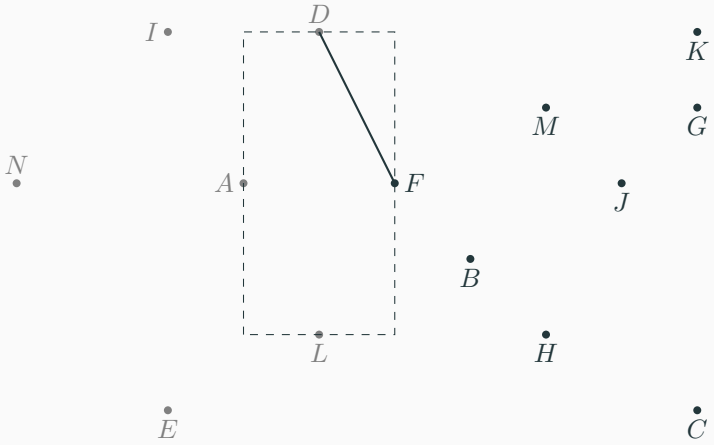
Visualização de identificação do par de pontos mais próximo

$\mathrm{dist}(F, A) = \mathrm{dist}(D, I) = \mathbf{2}$

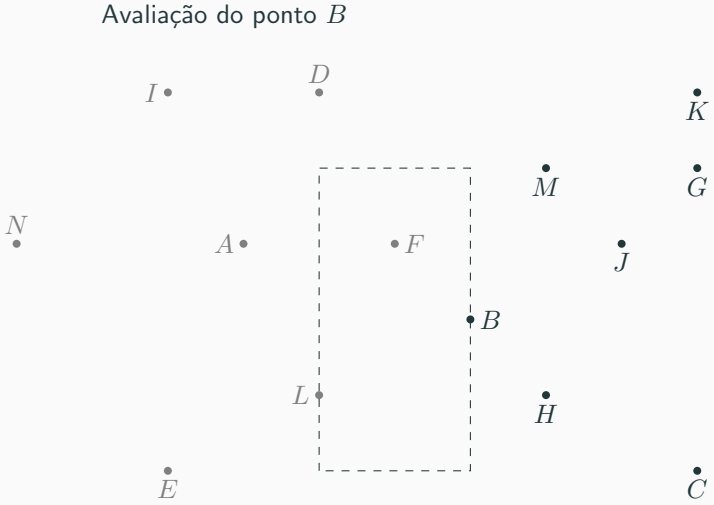


Visualização de identificação do par de pontos mais próximo

$$\text{dist}(F, D) = 2.236068 > \text{dist}(D, I) = \mathbf{2}$$

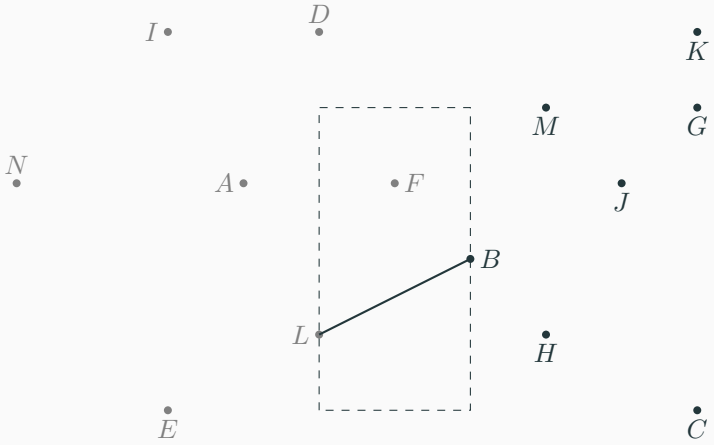


Visualização de identificação do par de pontos mais próximo



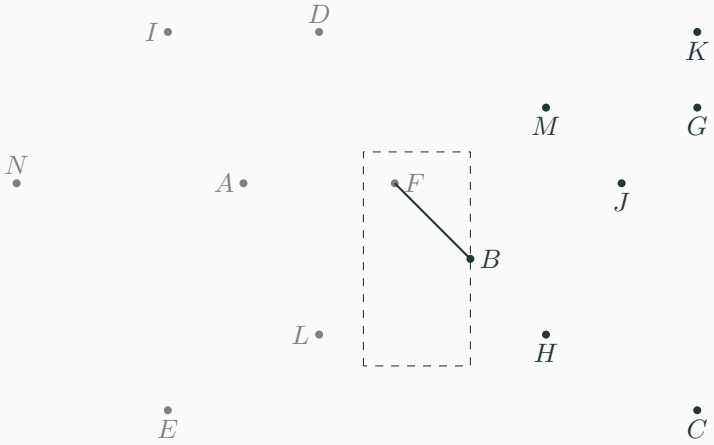
Visualização de identificação do par de pontos mais próximo

$$\text{dist}(B, L) = 2.236068 > \text{dist}(D, I) = \mathbf{2}$$



Visualização de identificação do par de pontos mais próximo

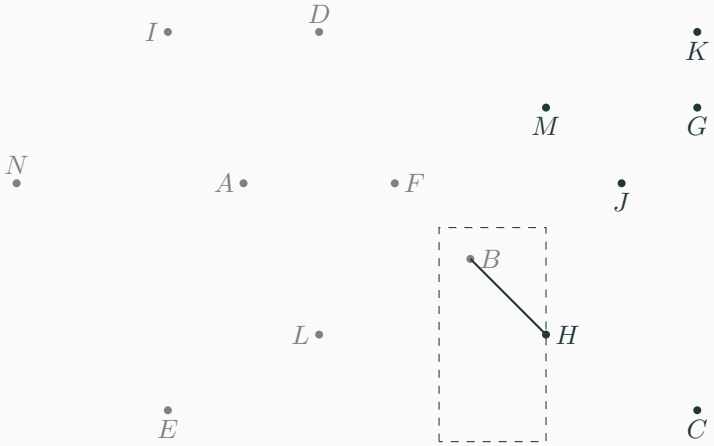
$$\mathrm{dist}(B, F) = \mathbf{1.414213} < \mathrm{dist}(D, I) = 2$$



Visualização de identificação do par de pontos mais próximo

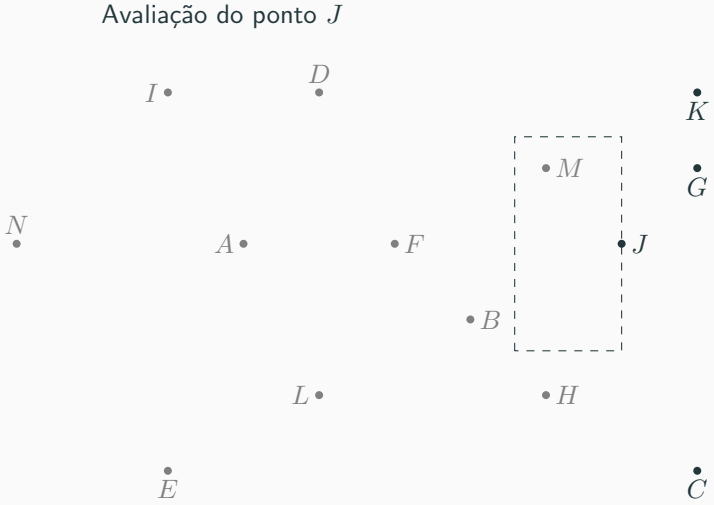
Visualização de identificação do par de pontos mais próximo

$$\mathrm{dist}(H, B) = \mathrm{dist}(B, F) = \mathbf{1.414213}$$



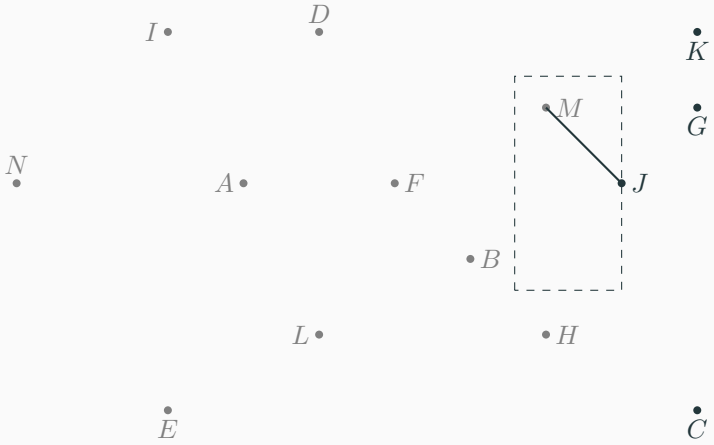
Visualização de identificação do par de pontos mais próximo

Visualização de identificação do par de pontos mais próximo



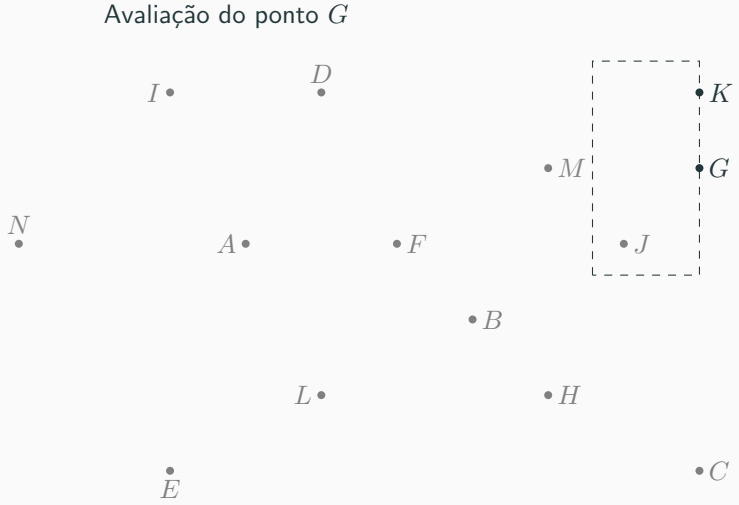
Visualização de identificação do par de pontos mais próximo

$$\mathrm{dist}(J, M) = \mathrm{dist}(B, F) = \mathbf{1.414213}$$



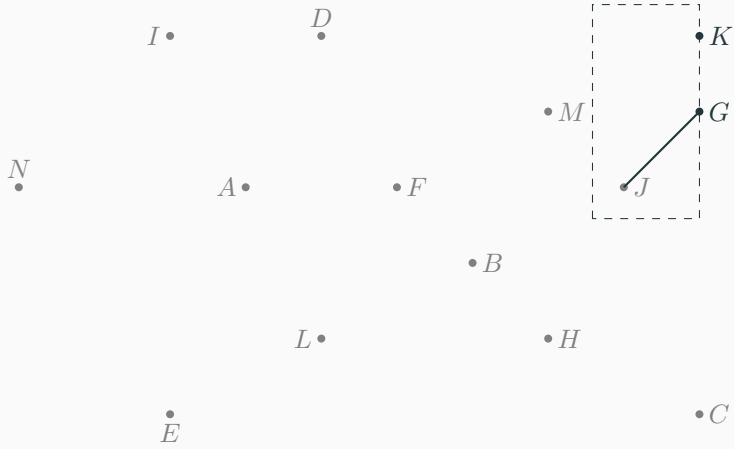
Visualização de identificação do par de pontos mais próximo

Visualização de identificação do par de pontos mais próximo



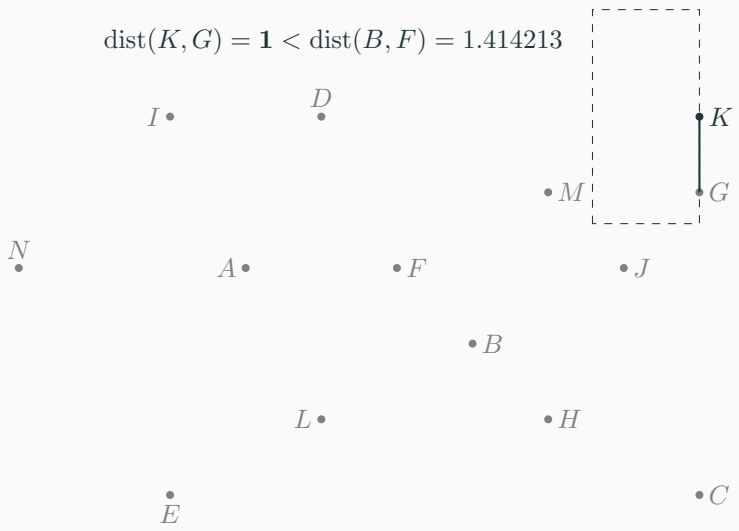
Visualização de identificação do par de pontos mais próximo

$$\mathrm{dist}(G, J) = \mathrm{dist}(B, F) = \mathbf{1.414213}$$



Visualização de identificação do par de pontos mais próximo

Visualização de identificação do par de pontos mais próximo



Implementação da identifacação do par mais próximo

```cpp
#include <bits/stdc++.h>

using namespace std;

struct Point {
    double x, y;

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

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

// Este código assume que N > 1
auto closest_pair(int N, vector<Point>& ps)
{
    sort(ps.begin(), ps.end());
```

Implementação da identifacação do par mais próximo

```
21    auto d = dist(ps[0], ps[1]);
22    auto closest = make_pair(ps[0], ps[1]);
23
24    set<Point> S;
25    S.emplace(ps[0].y, ps[0].x);
26    S.emplace(ps[1].y, ps[1].x);
27
28    for (int i = 2; i < N; ++i) {
29        auto P = ps[i];
30        auto it = S.lower_bound(Point(P.y - d, 0));
31
32        while (it != S.end()) {
33            auto Q = Point(it->y, it->x);
34
35            if (Q.x < P.x - d) {
36                it = S.erase(it);
37                continue;
38            }
```

Implementação da identifacação do par mais próximo

```
40              if (Q.y > P.y + d)
41                  break;
42
43              auto t = dist(P, Q);
44
45              if (t < d)
46              {
47                  d = t;
48                  closest = make_pair(P, Q);
49              }
50
51              ++it;
52          }
53
54          S.emplace(P.y, P.x);
55      }
56
57      return closest;
58  }
59
60  int main()
```

Interseção de segmentos de reta

Problema

- O problema da interseção de segmentos de reta consiste em determinar se, em um conjunto S composto por N segmentos de reta, existe um par de segmentos $r, s \in S$ tal que $r \cap s \neq \emptyset$

Problema

- O problema da interseção de segmentos de reta consiste em determinar se, em um conjunto S composto por N segmentos de reta, existe um par de segmentos $r, s \in S$ tal que $r \cap s \neq \emptyset$
- Uma variante comum é determinar todos os pontos de interseção entre estes segmentos

Problema

- O problema da interseção de segmentos de reta consiste em determinar se, em um conjunto S composto por N segmentos de reta, existe um par de segmentos $r, s \in S$ tal que $r \cap s \neq \emptyset$
- Uma variante comum é determinar todos os pontos de interseção entre estes segmentos
- A solução de busca completa testa cada elemento de S contra todos os demais

Problema

- O problema da interseção de segmentos de reta consiste em determinar se, em um conjunto S composto por N segmentos de reta, existe um par de segmentos $r, s \in S$ tal que $r \cap s \neq \emptyset$
- Uma variante comum é determinar todos os pontos de interseção entre estes segmentos
- A solução de busca completa testa cada elemento de S contra todos os demais
- Como a interseção entre dois segmentos pode ser obtida em $O(1)$ e existem $N(N-1)/2$ pares de segmentos distintos possíveis, esta abordagem tem complexidade $O(N^2)$

Problema

- O problema da interseção de segmentos de reta consiste em determinar se, em um conjunto S composto por N segmentos de reta, existe um par de segmentos $r, s \in S$ tal que $r \cap s \neq \emptyset$
- Uma variante comum é determinar todos os pontos de interseção entre estes segmentos
- A solução de busca completa testa cada elemento de S contra todos os demais
- Como a interseção entre dois segmentos pode ser obtida em $O(1)$ e existem $N(N-1)/2$ pares de segmentos distintos possíveis, esta abordagem tem complexidade $O(N^2)$
- Existe um algoritmo com menor complexidade para o problema apresentado, e algoritmos sensíveis à entrada para a variante

Algoritmo de Shamos-Hoey

- Shamos e Hoey propuseram, em 1976, um algoritmo capaz de determinar se existe ao menos uma interseção entre N segmentos de reta com complexidade $O(N \log N)$ e memória $O(N)$

Algoritmo de Shamos-Hoey

- Shamos e Hoey propuseram, em 1976, um algoritmo capaz de determinar se existe ao menos uma interseção entre N segmentos de reta com complexidade $O(N \log N)$ e memória $O(N)$
- A ideia é ordenar os N segmentos do conjunto S em ordem lexicográfica e manter uma árvore binária balanceada A de segmentos ativos

Algoritmo de Shamos-Hoey

- Shamos e Hoey propuseram, em 1976, um algoritmo capaz de determinar se existe ao menos uma interseção entre N segmentos de reta com complexidade $O(N \log N)$ e memória $O(N)$
- A ideia é ordenar os N segmentos do conjunto S em ordem lexicográfica e manter uma árvore binária balanceada A de segmentos ativos
- Cada segmento gera dois eventos: o ponto inicial do segmento gera um evento de inclusão de intervalo (1) e o ponto final do segmento um evento de exclusão do intervalo (2)

Algoritmo de Shamos-Hoey

- Shamos e Hoey propuseram, em 1976, um algoritmo capaz de determinar se existe ao menos uma interseção entre N segmentos de reta com complexidade $O(N \log N)$ e memória $O(N)$
- A ideia é ordenar os N segmentos do conjunto S em ordem lexicográfica e manter uma árvore binária balanceada A de segmentos ativos
- Cada segmento gera dois eventos: o ponto inicial do segmento gera um evento de inclusão de intervalo (1) e o ponto final do segmento um evento de exclusão do intervalo (2)
- A fila dos eventos deve ser ordenadas pelo ponto $P = (x_e, y_e)$ que deu origem ao evento

Algoritmo de Shamos-Hoey

- Shamos e Hoey propuseram, em 1976, um algoritmo capaz de determinar se existe ao menos uma interseção entre N segmentos de reta com complexidade $O(N \log N)$ e memória $O(N)$
- A ideia é ordenar os N segmentos do conjunto S em ordem lexicográfica e manter uma árvore binária balanceada A de segmentos ativos
- Cada segmento gera dois eventos: o ponto inicial do segmento gera um evento de inclusão de intervalo (1) e o ponto final do segmento um evento de exclusão do intervalo (2)
- A fila dos eventos deve ser ordenadas pelo ponto $P = (x_e, y_e)$ que deu origem ao evento
- Para cada evento, a árvore de segmentos ativos A deve estar ordenada pela coordenada y dos pontos dos segmentos com coordenada $x = x_e$

Algoritmo de Shamos-Hoey

- Para manter esta ordenação é necessário utilizar uma árvore binária de busca balanceada

Algoritmo de Shamos-Hoey

- Para manter esta ordenação é necessário utilizar uma árvore binária de busca balanceada
- Uma alternativa é implementar tal árvore (por exemplo, uma árvore *red-black*)

Algoritmo de Shamos-Hoey

- Para manter esta ordenação é necessário utilizar uma árvore binária de busca balanceada
- Uma alternativa é implementar tal árvore (por exemplo, uma árvore *red-black*)
- Outra alternativa é utilizar um `set` da linguagem C++, em conjunto com uma variável global que armazene o valor da coordenada x do evento atual e que seja utilizada na rotina de comparação

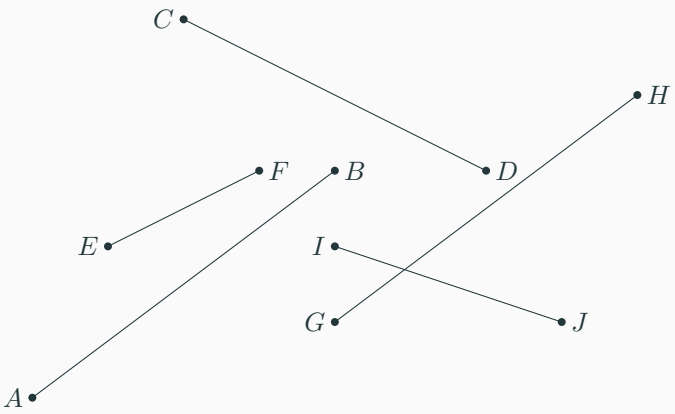
Algoritmo de Shamos-Hoey

- Para manter esta ordenação é necessário utilizar uma árvore binária de busca balanceada
- Uma alternativa é implementar tal árvore (por exemplo, uma árvore *red-black*)
- Outra alternativa é utilizar um `set` da linguagem C++, em conjunto com uma variável global que armazene o valor da coordenada x do evento atual e que seja utilizada na rotina de comparação
- Observe que, uma vez que um segmento r esteja abaixo de um outro segmento s em um ponto x, esta relação só mudará para valores maiores do que x caso exista uma interseção ambos

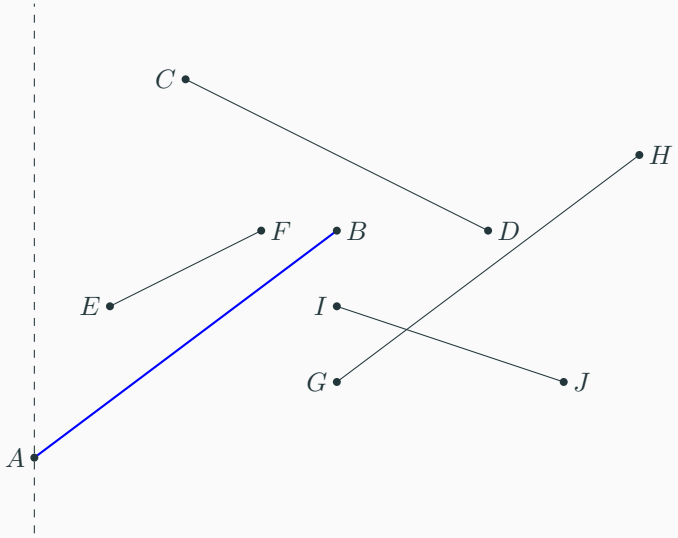
Algoritmo de Shamos-Hoey

- Para manter esta ordenação é necessário utilizar uma árvore binária de busca balanceada
- Uma alternativa é implementar tal árvore (por exemplo, uma árvore *red-black*)
- Outra alternativa é utilizar um `set` da linguagem C++, em conjunto com uma variável global que armazene o valor da coordenada x do evento atual e que seja utilizada na rotina de comparação
- Observe que, uma vez que um segmento r esteja abaixo de um outro segmento s em um ponto x, esta relação só mudará para valores maiores do que x caso exista uma interseção ambos
- No caso do algoritmo de Shamos-Hoey a existência de interseção é um critério de parada, logo não há necessidade de tratar tais casos

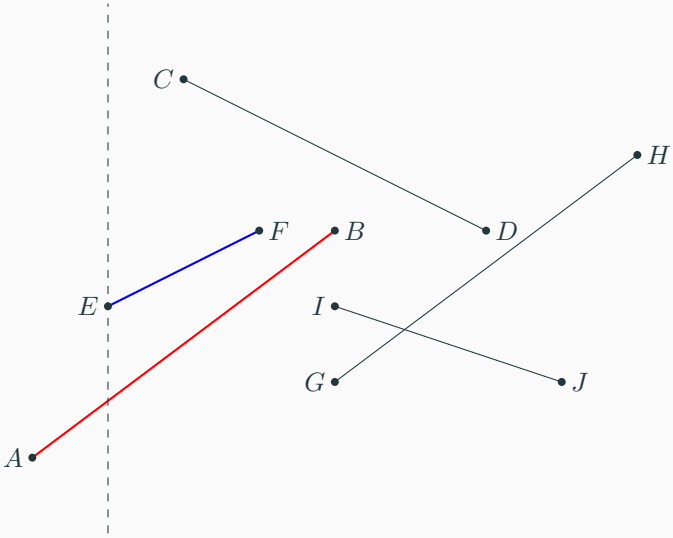
Visualização de identificação de interseção de segmentos



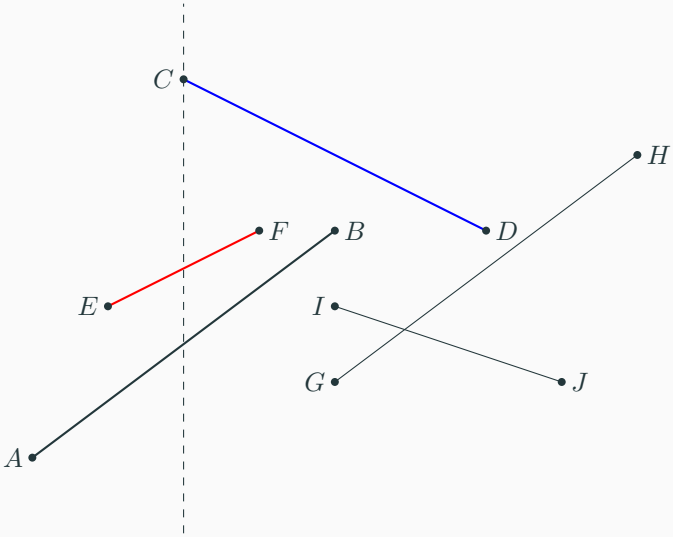
Visualização de identificação de interseção de segmentos



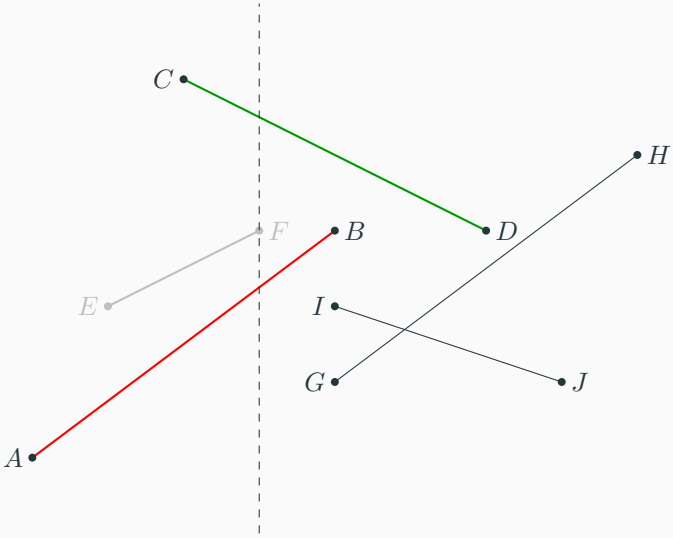
Visualização de identificação de interseção de segmentos



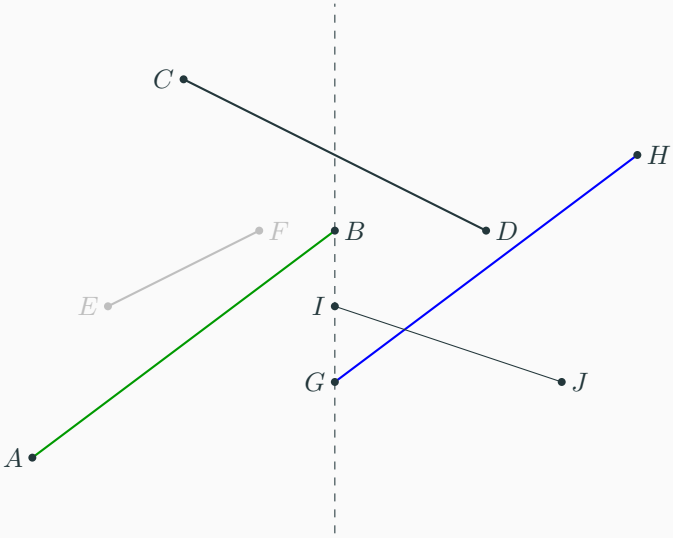
Visualização de identificação de interseção de segmentos



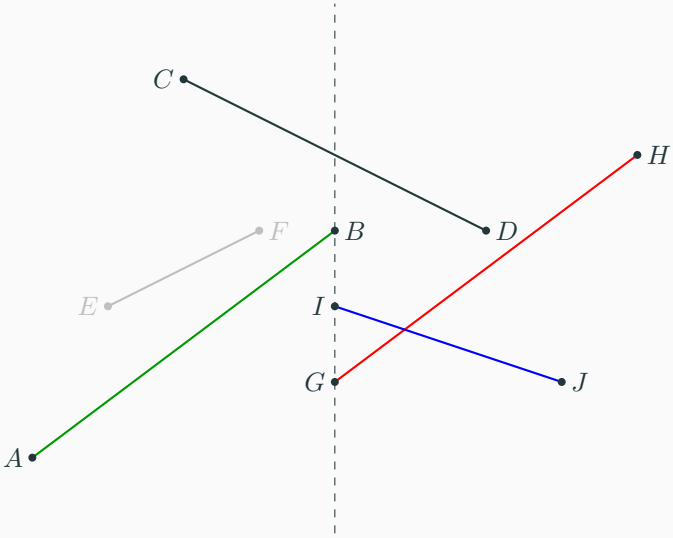
Visualização de identificação de interseção de segmentos



Visualização de identificação de interseção de segmentos



Visualização de identificação de interseção de segmentos



Implementação do algoritmo de Shamos-Hoey

```cpp
#include <bits/stdc++.h>

using namespace std;
using ll = long long;

template<typename T>
bool equals(T a, T b)
{
    if (std::is_floating_point<T>::value)
    {
        const double EPS { 1e-6 };

        return fabs(a - b) < EPS;
    } else
        return a == b;
}
```

Implementação do algoritmo de Shamos-Hoey

```
18 template<typename T>
19 struct Point
20 {
21     T x, y;
22
23     bool operator<(const Point& P) const
24     {
25         return x != P.x ? x < P.x : y < P.y;
26     }
27
28     bool operator==(const Point& P) const { return x == P.x and y == P.y; }
29 };
30
31 template<typename T>
32 T D(const Point<T>& P, const Point<T>& Q, const Point<T>& R)
33 {
34     return (P.x*Q.y + P.y*R.x + Q.x*R.y) - (R.x*Q.y + R.y*P.x + Q.x*P.y);
35 }
```

Implementação do algoritmo de Shamos-Hoey

```
37 template<typename T>
38 struct Segment
39 {
40     T a, b, c;
41     Point<T> A, B;
42
43     Segment(const Point<T>& P, const Point<T>& Q)
44         : a(P.y - Q.y), b (Q.x - P.x), c(P.x*Q.y - Q.x*P.y), A(P), B(Q) { sweep_x = -1; }
45
46     bool operator<(const Segment& line) const
47     {
48         return (-a*sweep_x - c)*line.b < (-line.a*sweep_x -line.c)*b;
49     }
50
51     bool intersect(const Segment& s) const
52     {
53         auto d1 = D(A, B, s.A), d2 = D(A, B, s.B);
54
55         if ((equals(d1, 0LL) && contains(s.A)) || (equals(d2, 0LL) && contains(s.B)))
56             return true;
```

Implementação do algoritmo de Shamos-Hoey

```
58          auto d3 = D(s.A, s.B, A);
59          auto d4 = D(s.A, s.B, B);
60
61          if ((equals(d3, 0LL) && s.contains(A)) || (equals(d4, 0LL) && s.contains(B)))
62              return true;
63
64          return (d1 * d2 < 0) && (d3 * d4 < 0);
65      }
66
67      bool contains(const Point<T>& P) const
68      {
69          if (P == A || P == B)
70              return true;
71
72          auto xmin = min(A.x, B.x);
73          auto xmax = max(A.x, B.x);
74          auto ymin = min(A.y, B.y);
75          auto ymax = max(A.y, B.y);
```

Implementação do algoritmo de Shamos-Hoey

```
77          if (P.x < xmin || P.x > xmax || P.y < ymin || P.y > ymax)
78              return false;
79
80          return equals((P.y - A.y)*(B.x - A.x), (P.x - A.x)*(B.y - A.y));
81      }
82
83      static T sweep_x;
84  };
85
86  template<typename T>
87  T Segment<T>::sweep_x;
88
89  template<typename T>
90  bool shamos_hoey(const vector<Segment<T>>& segments)
91  {
92      struct Event
93      {
94          Point<T> P;
95          size_t i;
```

Implementação do algoritmo de Shamos-Hoey

```
97          bool operator<(const Event& e) const { return P < e.P; }
98      };
99
100     vector<Event> events;
101
102     for (size_t i = 0; i < segments.size(); ++i)
103     {
104         events.push_back({ segments[i].A, i });
105         events.push_back({ segments[i].B, i });
106     }
107
108     sort(events.begin(), events.end());
109     set<Segment<T>> sl;
110
111     for (const auto& e : events)
112     {
113         auto s = segments[e.i];
114         Segment<T>::sweep_x = e.P.x;
```

Implementação do algoritmo de Shamos-Hoey

```
116         if (e.P == s.A)
117         {
118             sl.insert(s);
119
120             auto it = sl.find(s);
121
122             if (it != sl.begin())
123             {
124                 auto L = *prev(it);
125
126                 if (s.intersect(L)) return true;
127             }
128
129             if (next(it) != sl.end())
130             {
131                 auto U = *next(it);
132
133                 if (s.intersect(U)) return true;
134             }
```

Implementação do algoritmo de Shamos-Hoey

```
135         } else
136         {
137             auto it = sl.find(s);
138 
139             if (it != sl.begin() and it != sl.end())
140             {
141                 auto L = *prev(it);
142                 auto U = *next(it);
143 
144                 if (L.intersect(U)) return true;
145             }
146 
147             sl.erase(it);
148         }
149     }
150 
151     return false;
152 }
```

Algoritmo de Bentley-Ottman

- O algoritmo de Bentley-Ottman é uma extensão do algoritmo de Shamos-Hoey que permite identificação todos os pontos de interseção entre os segmentos

Algoritmo de Bentley-Ottman

- O algoritmo de Bentley-Ottman é uma extensão do algoritmo de Shamos-Hoey que permite identificação todos os pontos de interseção entre os segmentos
- A complexidade do algoritmo é $O((N + k) \log N)$, onde k é o número de pontos de interseção entre os segmentos

Algoritmo de Bentley-Ottman

- O algoritmo de Bentley-Ottman é uma extensão do algoritmo de Shamos-Hoey que permite identificação todos os pontos de interseção entre os segmentos
- A complexidade do algoritmo é $O((N + k) \log N)$, onde k é o número de pontos de interseção entre os segmentos
- Como o número máximo de intercessões k entre N segmentos é $O(N^2)$, no pior caso o algoritmo de Bentley-Ottman tem complexidade pior do que a busca completa

Algoritmo de Bentley-Ottman

- O algoritmo de Bentley-Ottman é uma extensão do algoritmo de Shamos-Hoey que permite identificação todos os pontos de interseção entre os segmentos
- A complexidade do algoritmo é $O((N + k) \log N)$, onde k é o número de pontos de interseção entre os segmentos
- Como o número máximo de intercessões k entre N segmentos é $O(N^2)$, no pior caso o algoritmo de Bentley-Ottman tem complexidade pior do que a busca completa
- Este é um algoritmo sensível à entrada, pois sua complexidade depende de k

Implementação do algoritmo de busca completa

```cpp
#include <bits/stdc++.h>

using namespace std;
using ll = long long;

template<typename T>
bool equals(T a, T b)
{
    if (std::is_floating_point<T>::value)
    {
        const double EPS { 1e-6 };

        return fabs(a - b) < EPS;
    } else
        return a == b;
}
```

Implementação do algoritmo de busca completa

```
18 template<typename T>
19 struct Point
20 {
21     T x, y;
22
23     bool operator<(const Point& P) const
24     {
25         return x != P.x ? x < P.x : y < P.y;
26     }
27
28     bool operator==(const Point& P) const { return x == P.x and y == P.y; }
29 };
30
31 template<typename T>
32 struct Segment
33 {
34     T a, b, c;
35     Point<T> A, B;
```

Implementação do algoritmo de busca completa

```
37      Segment(const Point<T>& P, const Point<T>& Q)
38          : a(P.y - Q.y), b (Q.x - P.x), c(P.x*Q.y - Q.x*P.y), A(P), B(Q) { }
39
40      optional<Point<T>> intersection(const Segment& s)
41      {
42          auto det = a * s.b - b * s.a;
43
44          if (not equals(det, 0.0))     // Concorrentes
45          {
46              auto x = (-c * s.b + s.c * b) / det;
47              auto y = (-s.c * a + c * s.a) / det;
48
49              if (min(A.x, B.x) <= x and x <= max(A.x, B.x) and
50                  min(s.A.x, s.B.x) <= x and x <= max(s.A.x, s.B.x))
51              {
52                  return Point<T> { x, y };
53              }
54          }
55
56          return { };
```

Implementação do algoritmo de busca completa

```
60 template<typename T>
61 set<Point<T>> intersections(int N, const vector<Segment<T>>& segments)
62 {
63     set<Point<T>> ans;
64
65     for (int i = 0; i < N; ++i)
66     {
67         auto s = segments[i];
68
69         for (int j = i + 1; j < N; ++j)
70         {
71             auto r = segments[j];
72             auto P = s.intersection(r);
73
74             if (P) ans.insert(P.value());
75         }
76     }
77
78     return ans;
79 }
```

Algoritmo de Bentley-Ottman

- A estrutura geral do algoritmo de Bentley-Ottman é a mesma do algoritmos de Shamos-Hoey

Algoritmo de Bentley-Ottman

- A estrutura geral do algoritmo de Bentley-Ottman é a mesma do algoritmos de Shamos-Hoey
- A principal diferença é que os pontos de interseção entre os segmentos geram novos eventos

Algoritmo de Bentley-Ottman

- A estrutura geral do algoritmo de Bentley-Ottman é a mesma do algoritmos de Shamos-Hoey
- A principal diferença é que os pontos de interseção entre os segmentos geram novos eventos
- Em um evento de interseção, os segmentos que se interceptaram devem trocar de posições

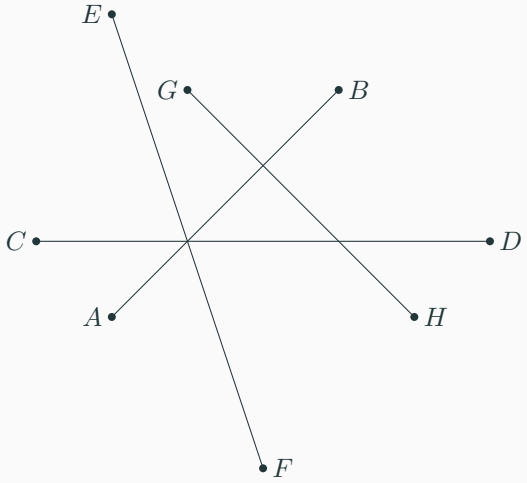
Algoritmo de Bentley-Ottman

- A estrutura geral do algoritmo de Bentley-Ottman é a mesma do algoritmos de Shamos-Hoey
- A principal diferença é que os pontos de interseção entre os segmentos geram novos eventos
- Em um evento de interseção, os segmentos que se interceptaram devem trocar de posições
- Esta operação pode ser implementada em uma árvore binária de busca balanceada aumentada, o que aumenta o tamanho e a complexidade da implementação

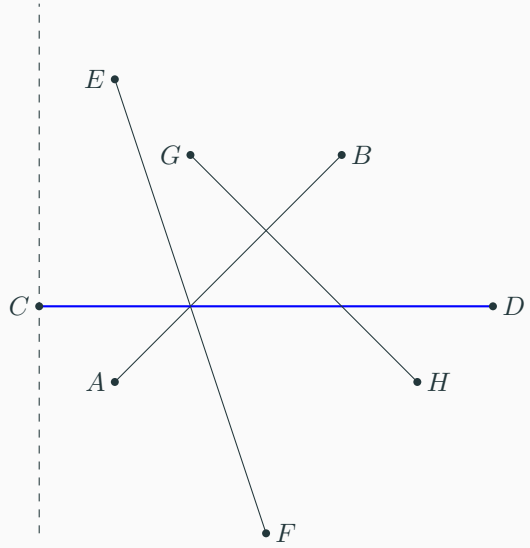
Algoritmo de Bentley-Ottman

- A estrutura geral do algoritmo de Bentley-Ottman é a mesma do algoritmos de Shamos-Hoey
- A principal diferença é que os pontos de interseção entre os segmentos geram novos eventos
- Em um evento de interseção, os segmentos que se interceptaram devem trocar de posições
- Esta operação pode ser implementada em uma árvore binária de busca balanceada aumentada, o que aumenta o tamanho e a complexidade da implementação
- Para usar o contêiner `set` da STL é preciso fazer algumas adaptações e assumir certas condições extras à entrada do problema

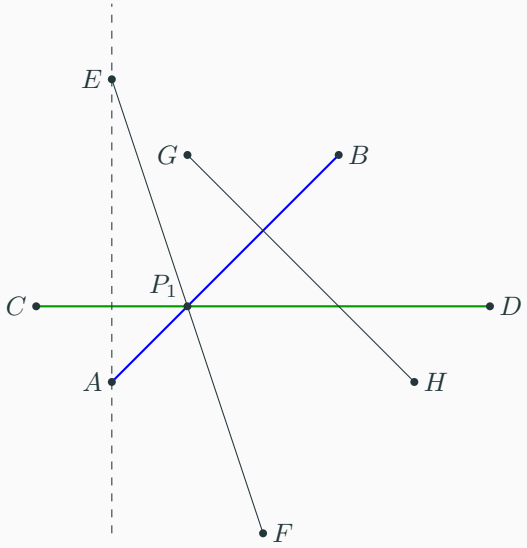
Visualização da contagem das interseções entres segmentos



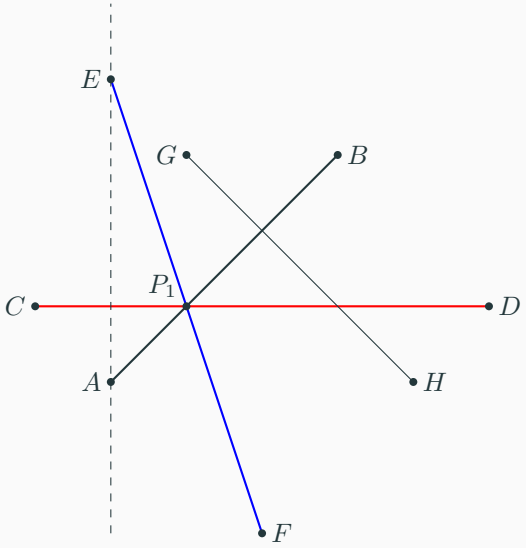
Visualização da contagem das interseções entres segmentos



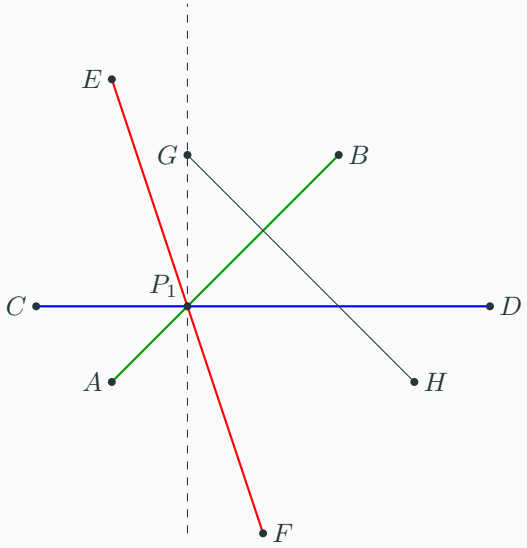
Visualização da contagem das interseções entres segmentos



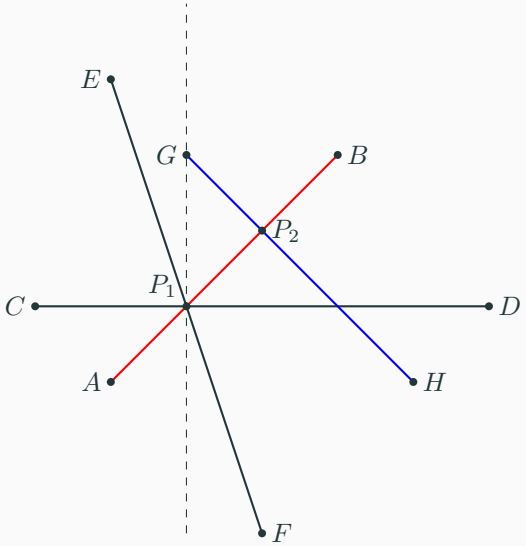
Visualização da contagem das interseções entres segmentos



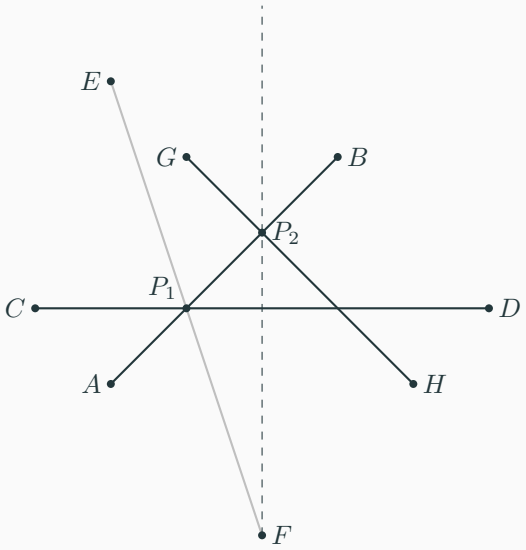
Visualização da contagem das interseções entres segmentos



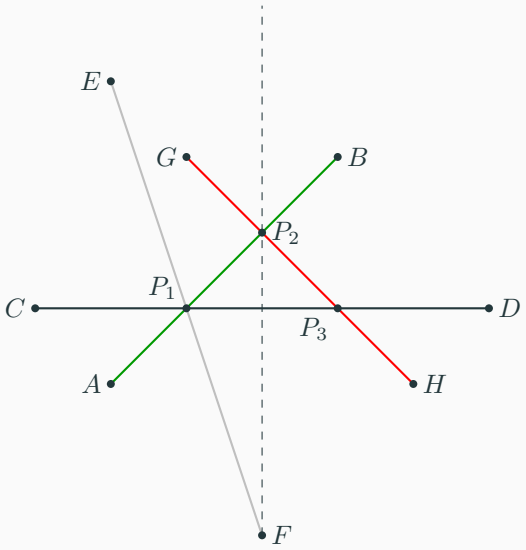
Visualização da contagem das interseções entres segmentos



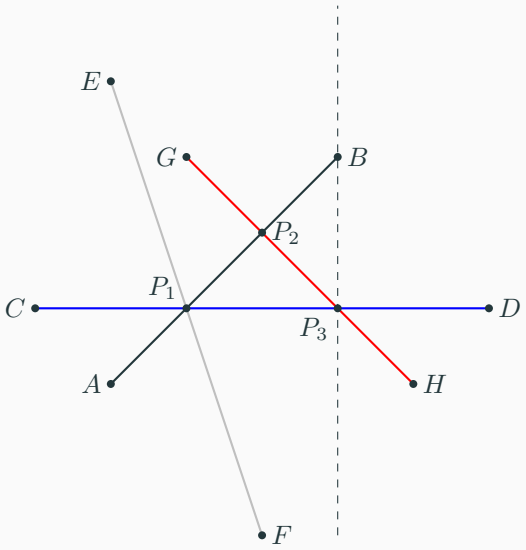
Visualização da contagem das interseções entres segmentos



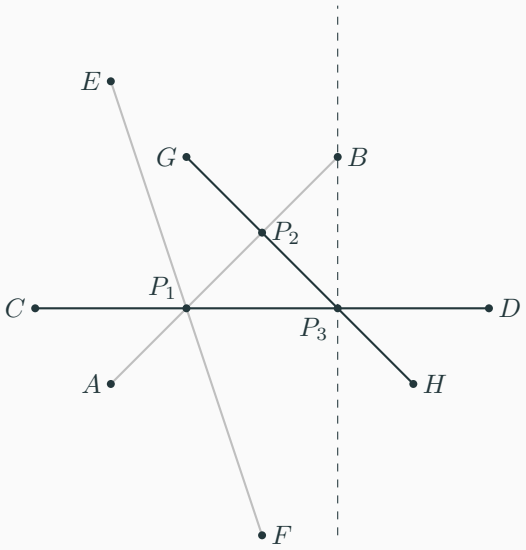
Visualização da contagem das interseções entres segmentos



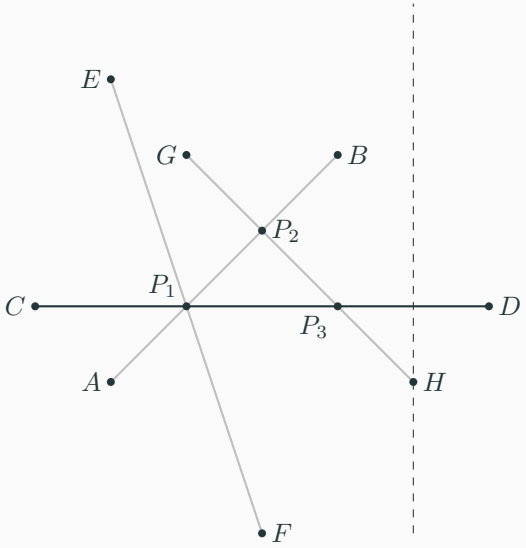
Visualização da contagem das interseções entres segmentos



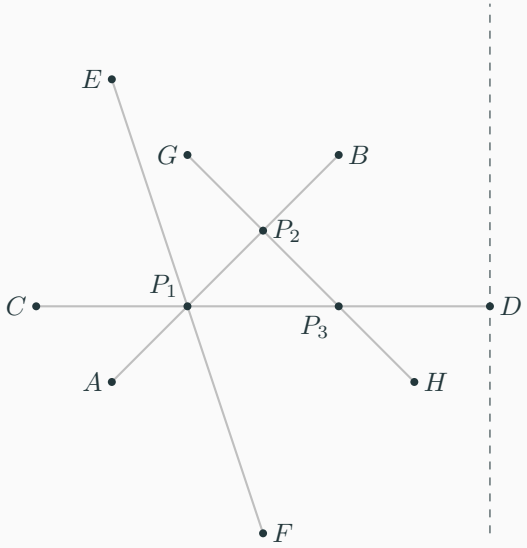
Visualização da contagem das interseções entres segmentos



Visualização da contagem das interseções entres segmentos



Visualização da contagem das interseções entres segmentos



Implementação do algoritmo de Bentley-Ottman

```cpp
#include <bits/stdc++.h>

using namespace std;

bool equals(double a, double b)
{
    const double EPS { 1e-6 };

    return fabs(a - b) < EPS;
}

struct Point
{
    double x, y;

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

Implementação do algoritmo de Bentley-Ottman

```
21      bool operator==(const Point& P) const
22      {
23          return x == P.x and y == P.y;
24      }
25
26      bool operator!=(const Point& P) const
27      {
28          return not (*this == P);
29      }
30  };
31
32  struct Segment
33  {
34      double a, b, c;
35      Point A, B;
36      size_t idx;
37
38      Segment(const Point& P, const Point& Q, size_t i)
39          : a(P.y - Q.y), b (Q.x - P.x), c(P.x*Q.y - Q.x*P.y), A(P), B(Q), idx(i) { }
```

Implementação do algoritmo de Bentley-Ottman

```
41      bool operator<(const Segment& s) const
42      {
43          return (-a*sweep_x - c)*s.b < (-s.a*sweep_x -s.c)*b;
44      }
45
46      optional<Point> intersection(const Segment& s) const
47      {
48          auto det = a * s.b - b * s.a;
49
50          if (not equals(det, 0.0))     // Concorrentes
51          {
52              auto x = (-c * s.b + s.c * b) / det;
53              auto y = (-s.c * a + c * s.a) / det;
54
55              if (min(A.x, B.x) <= x and x <= max(A.x, B.x) and
56                  min(s.A.x, s.B.x) <= x and x <= max(s.A.x, s.B.x))
57              {
58                  return Point { x, y };
59              }
60          }
```

Implementação do algoritmo de Bentley-Ottman

```
62          return { };
63      }
64
65      static double sweep_x;
66  };
67
68  double Segment::sweep_x;
69
70  struct Event
71  {
72      enum Type { OPEN, INTERSECTION, CLOSE };
73
74      Point P;
75      Type type;
76      size_t i;
77
78      bool operator<(const Event& e) const
79      {
80          if (P != e.P)
81              return e.P < P;
```

Implementação do algoritmo de Bentley-Ottman

```
83          if (type != e.type)
84              return type > e.type;
85
86          return i > e.i;
87      }
88  };
89
90  void add_neighbor_intersections(const Segment& s, const set<Segment>& sl,
91      set<Point>& ans, priority_queue<Event>& events)
92  {
93      // TODO: garantir que a busca identifique unicamente o elemento s,
94      // através do ajuste fino da variável Segment::sweep_x
95      auto it = sl.find(s);
96
97      if (it != sl.begin())
98      {
99          auto L = *prev(it);
100         auto P = s.intersection(L);
```

Implementação do algoritmo de Bentley-Ottman

```
102         if (P and ans.count(P.value()) == 0)
103         {
104             events.push(Event { P.value(), Event::INTERSECTION, s.idx } );
105             ans.insert(P.value());
106         }
107     }
108
109     if (next(it) != sl.end())
110     {
111         auto U = *next(it);
112         auto P = s.intersection(U);
113
114         if (P and ans.count(P.value()) == 0)
115         {
116             events.push(Event { P.value(), Event::INTERSECTION, s.idx } );
117             ans.insert(P.value());
118         }
119     }
120 }
```

Implementação do algoritmo de Bentley-Ottman

```
122 set<Point> bentley_ottman(vector<Segment>& segments)
123 {
124     set<Point> ans;
125     priority_queue<Event> events;
126 
127     for (size_t i = 0; i < segments.size(); ++i)
128     {
129         events.push(Event { segments[i].A, Event::OPEN, i });
130         events.push(Event { segments[i].B, Event::CLOSE, i });
131     }
132 
133     set<Segment> sl;
134 
135     while (not events.empty())
136     {
137         auto e = events.top();
138         events.pop();
139 
140         Segment::sweep_x = e.P.x;
```

Implementação do algoritmo de Bentley-Ottman

```
142         switch (e.type) {
143         case Event::OPEN:
144         {
145             auto s = segments[e.i];
146             sl.insert(s);
147 
148             add_neighbor_intersections(s, sl, ans, events);
149         }
150         break;
151 
152         case Event::CLOSE:
153         {
154             auto s = segments[e.i];
155             auto it = sl.find(s);       // TODO: aqui também
156 
157             if (it != sl.begin() and it != sl.end())
158             {
159                 auto L = *prev(it);
160                 auto U = *next(it);
161                 auto P = L.intersection(U);
```

Implementação do algoritmo de Bentley-Ottman

```
163                 if (P and ans.count(P.value()) == 0)
164                     events.push( Event { P.value(), Event::INTERSECTION, L.idx } );
165             }
166
167             sl.erase(it);
168         }
169         break;
170
171         default:
172             auto r = segments[e.i];
173             auto p = sl.equal_range(r);
174
175             vector<Segment> range(p.first, p.second);
176
177             // Remove os segmentos que se interceptam
178             sl.erase(p.first, p.second);
179
180             // Reinsere os segmentos
181             Segment::sweep_x += 0.1;
```

Implementação do algoritmo de Bentley-Ottman

```
183             sl.insert(range.begin(), range.end());
184
185             // Procura interseções com os novos vizinhos
186             for (const auto& s : range)
187                 add_neighbor_intersections(s, sl, ans, events);
188         }
189     }
190
191     return ans;
192 }
```

Referências

1. **De BERG**, Mark; **CHEONG**, Otfried. *Computational Geometry: Algorithms and Applications*, 2008.
2. **HALIM**, Felix; **HALIM**, Steve. *Competitive Programming 3*, 2010.
3. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, 2018.
4. **SUNDAY**, Dan. Intersections of a Set of Segments, acesso em 25/05/2019.
5. Wikipedia. Sweep line algorithm, acesso em 22/05/2019.