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

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

- 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

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

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

Par de pontos mais próximo

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

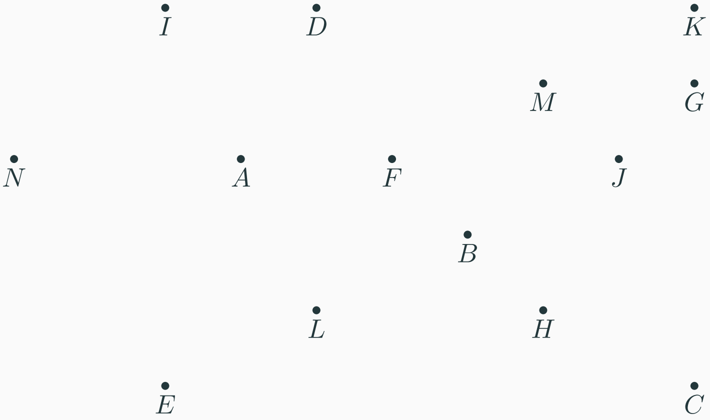
- Seja $d = \mathrm{dist}(P_1, P_2)$
- Agora, para todos pontos $P_3, P_4, \ldots, P_N$, deve-se computar todos os pontos vizinhos de $P_i = (x, y)$ tais que as coordenadas x estejam no intervalo $[x - d, x]$ e que as coordenadas y estejam no intervalo $[y - d, y + d]$
- Estes pontos podem ser identificados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

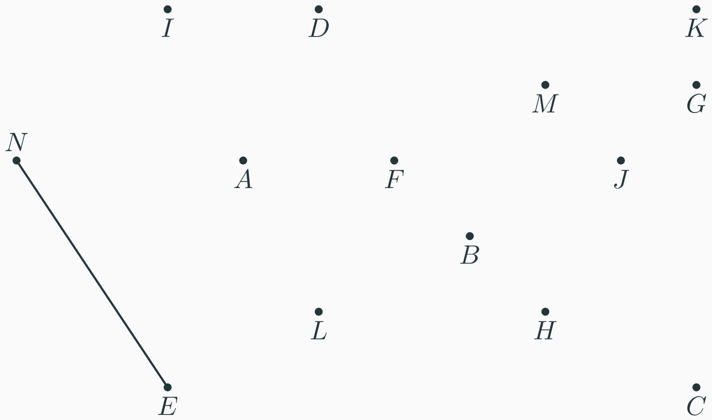
Visualização de identificação do par de pontos mais próximo

Par mais próximo: –



Visualização de identificação do par de pontos mais próximo

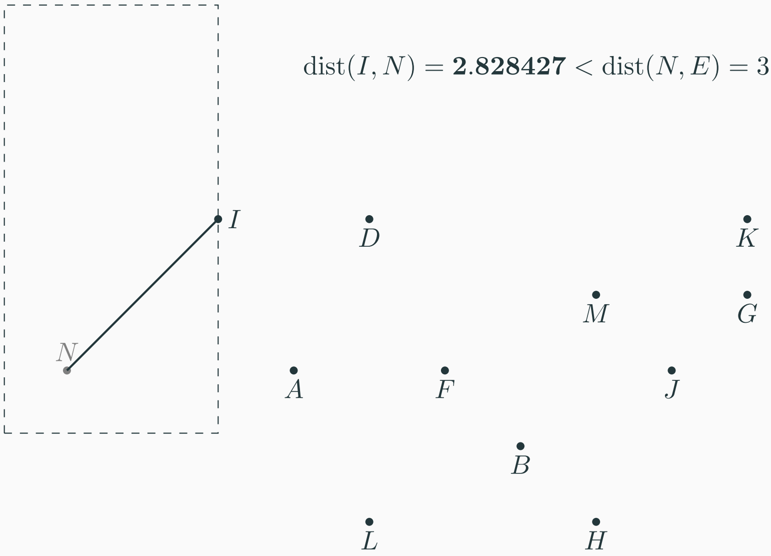
Par inicial, $\text{dist}(N, E) = \mathbf{3.605551}$



Visualização de identificação do par de pontos mais próximo

Visualização de identificação do par de pontos mais próximo

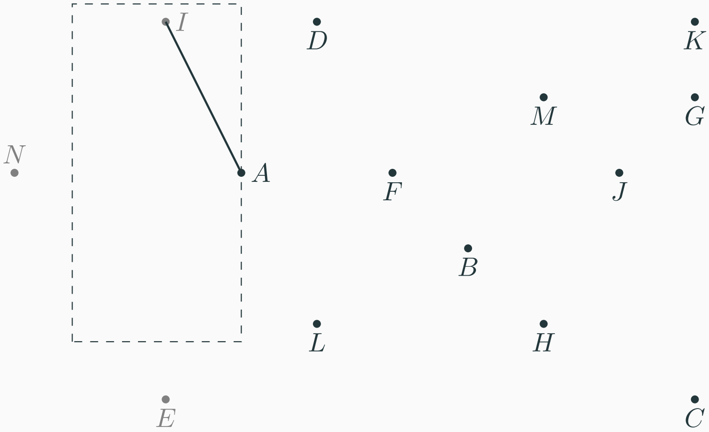
$$\text{dist}(I, N) = \mathbf{2.828427} < \text{dist}(N, E) = 3.605551$$



Visualização de identificação do par de pontos mais próximo

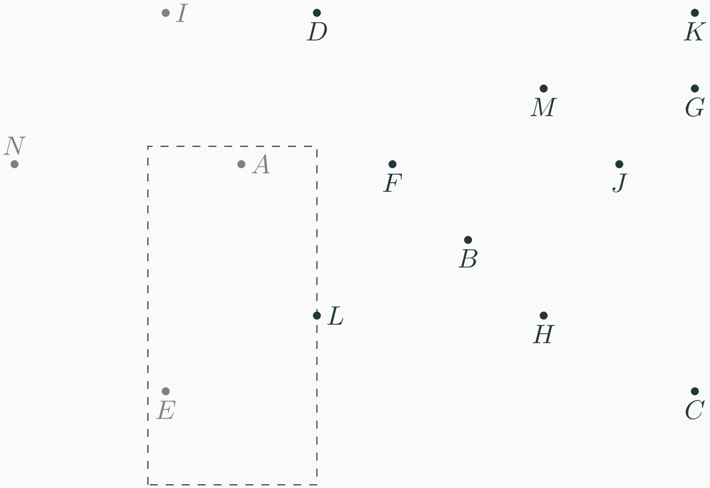
Visualização de identificação do par de pontos mais próximo

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



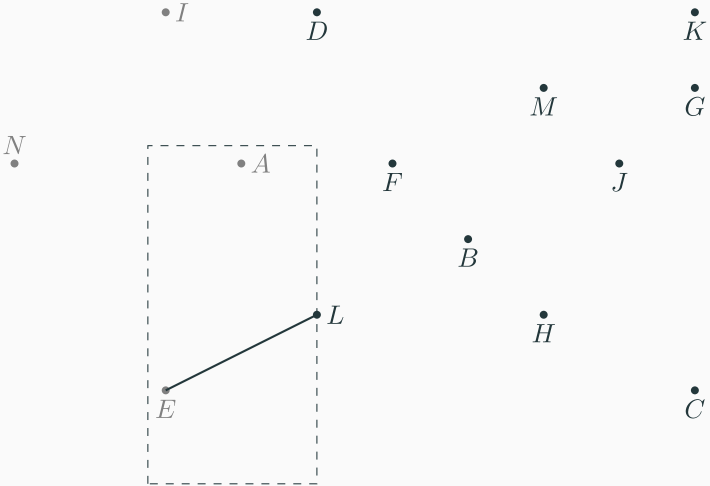
Visualização de identificação do par de pontos mais próximo

Avaliação do ponto L



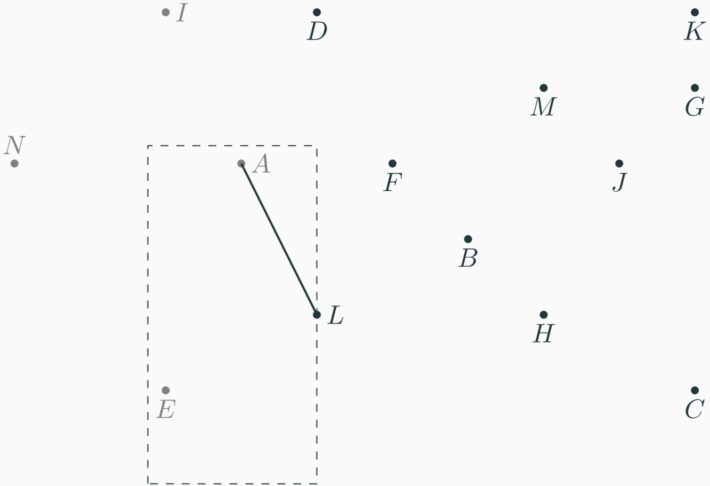
Visualização de identificação do par de pontos mais próximo

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



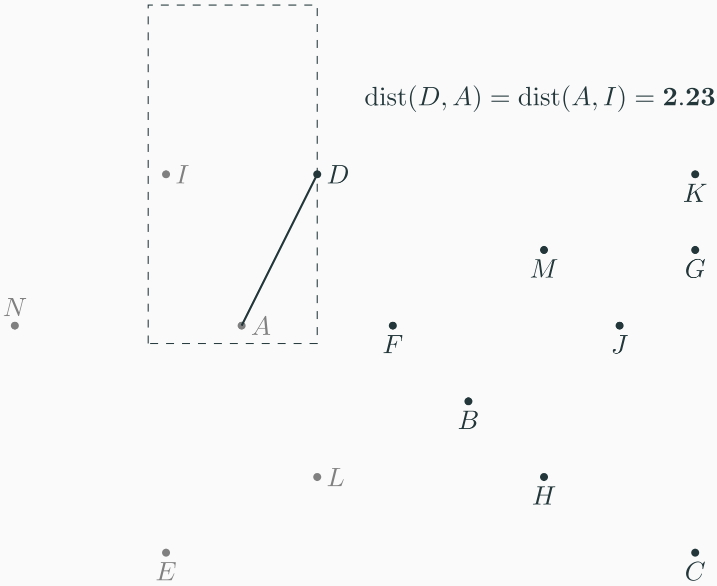
Visualização de identificação do par de pontos mais próximo

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



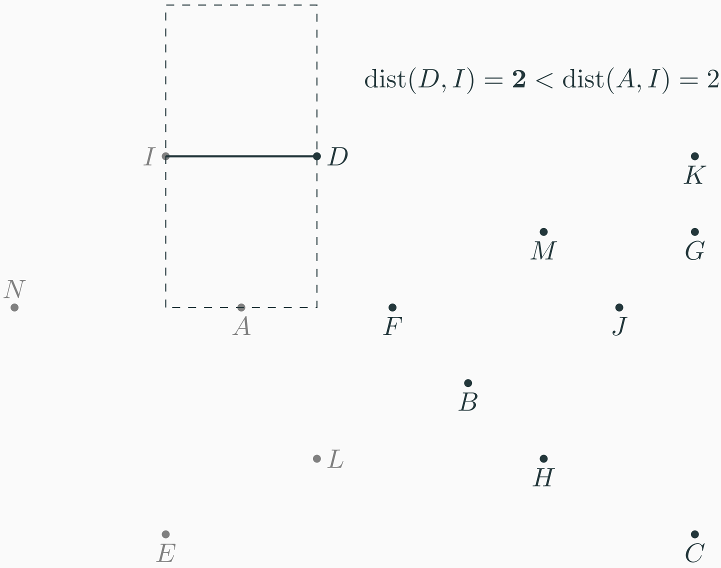
Visualização de identificação do par de pontos mais próximo

Visualização de identificação do par de pontos mais próximo



$$\text{dist}(D, A) = \text{dist}(A, I) = \mathbf{2.236068}$$

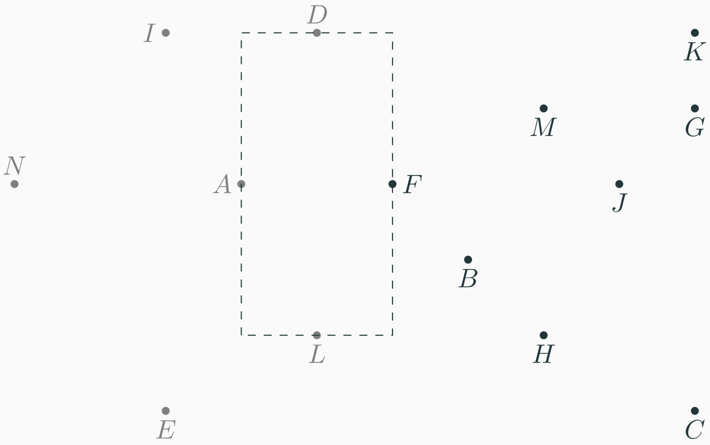
Visualização de identificação do par de pontos mais próximo



$$\mathrm{dist}(D, I) = \mathbf{2} < \mathrm{dist}(A, I) = 2.236068$$

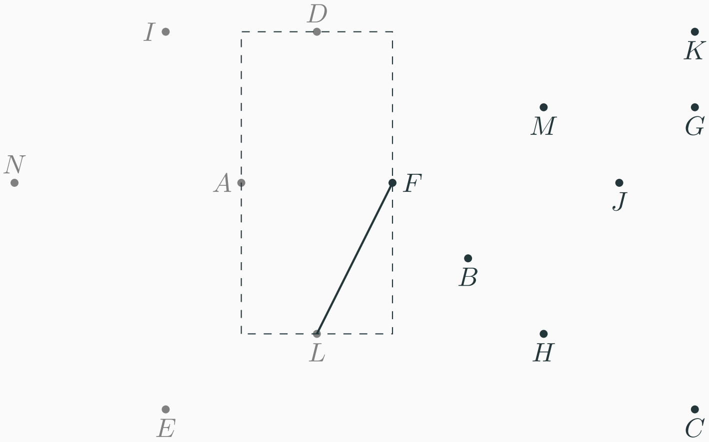
Visualização de identificação do par de pontos mais próximo

Avaliação do ponto F



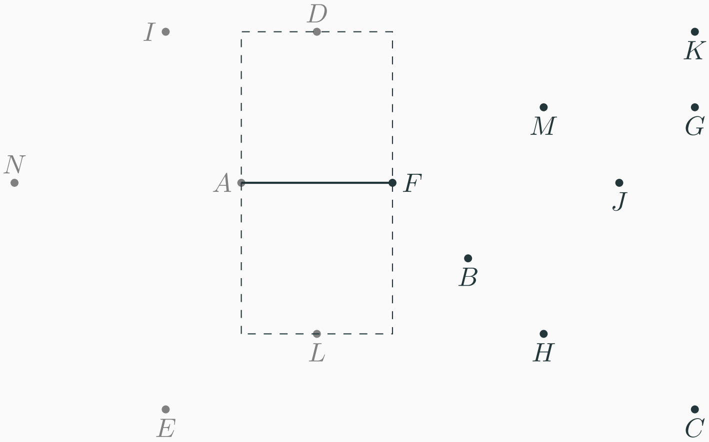
Visualização de identificação do par de pontos mais próximo

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



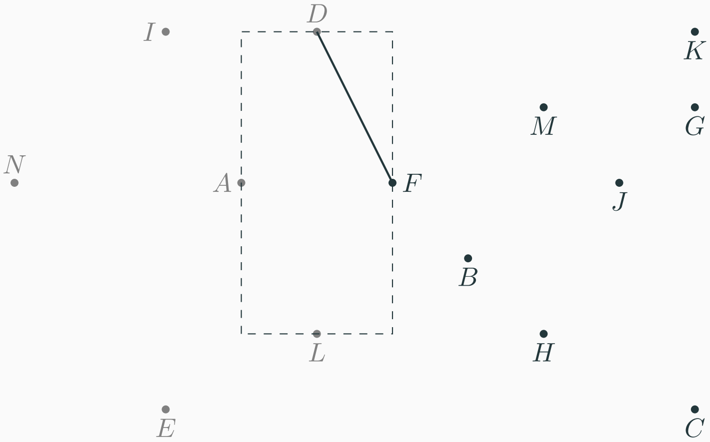
Visualização de identificação do par de pontos mais próximo

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



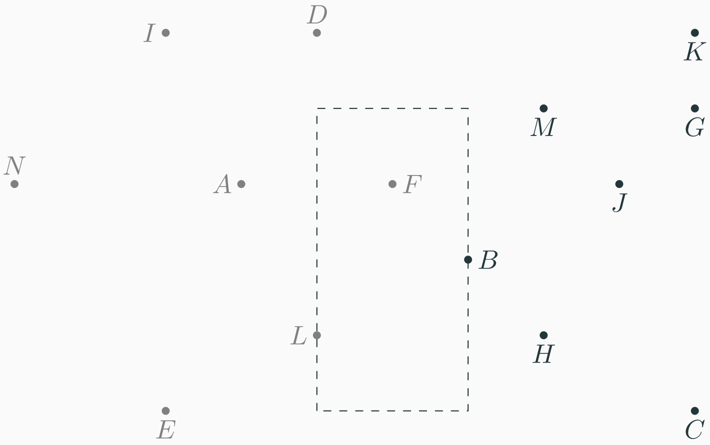
Visualização de identificação do par de pontos mais próximo

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



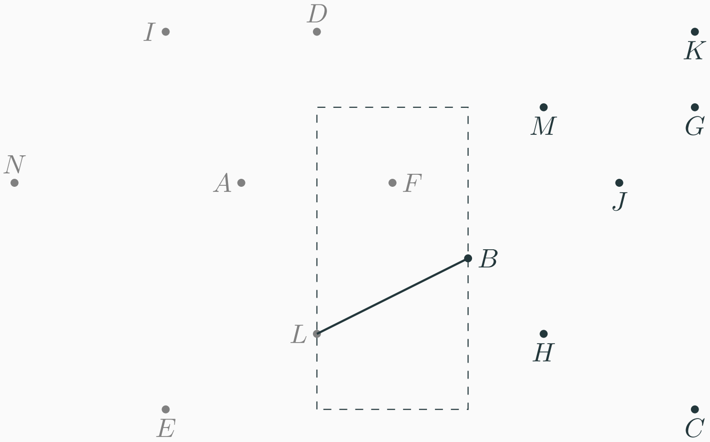
Visualização de identificação do par de pontos mais próximo

Avaliação do ponto B



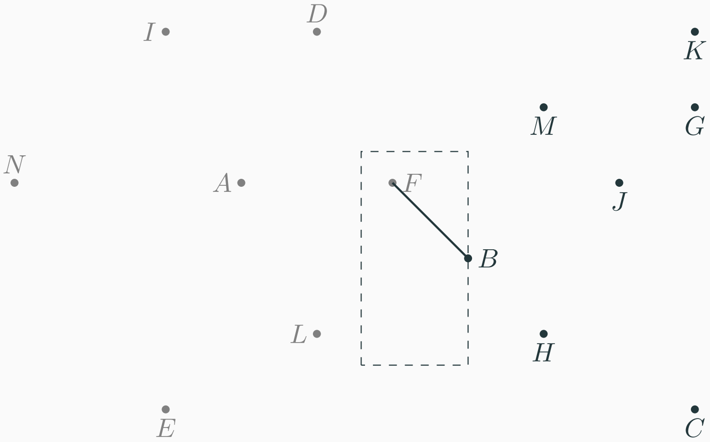
Visualização de identificação do par de pontos mais próximo

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



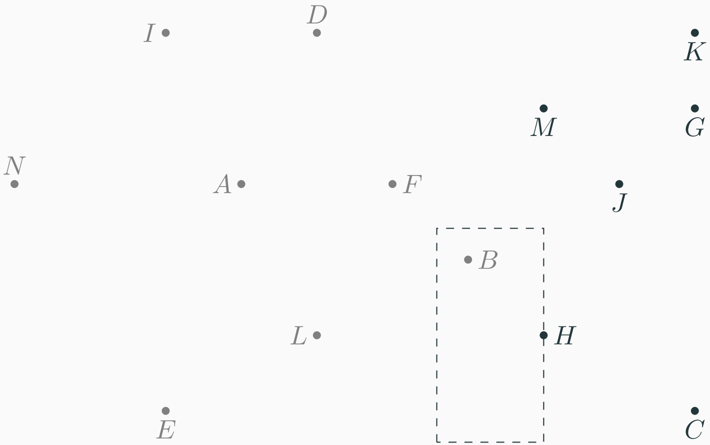
Visualização de identificação do par de pontos mais próximo

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



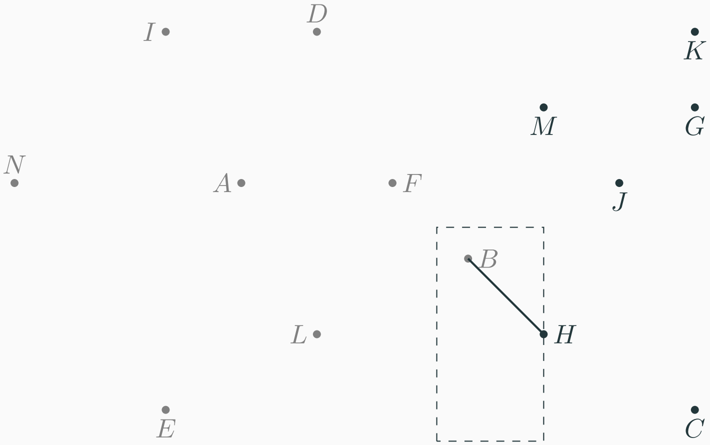
Visualização de identificação do par de pontos mais próximo

Avaliação do ponto H



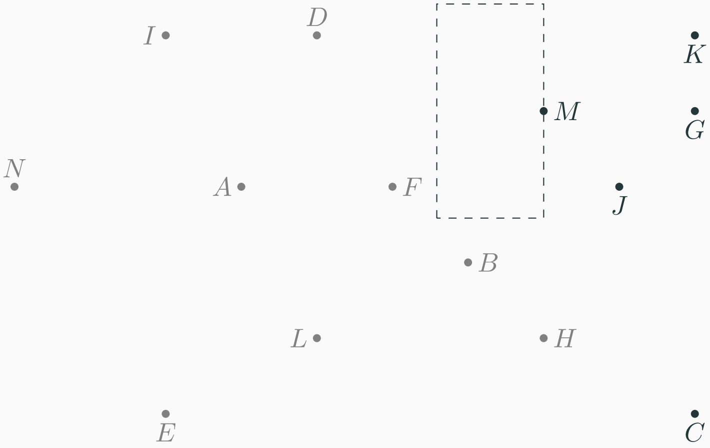
Visualização de identificação do par de pontos mais próximo

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



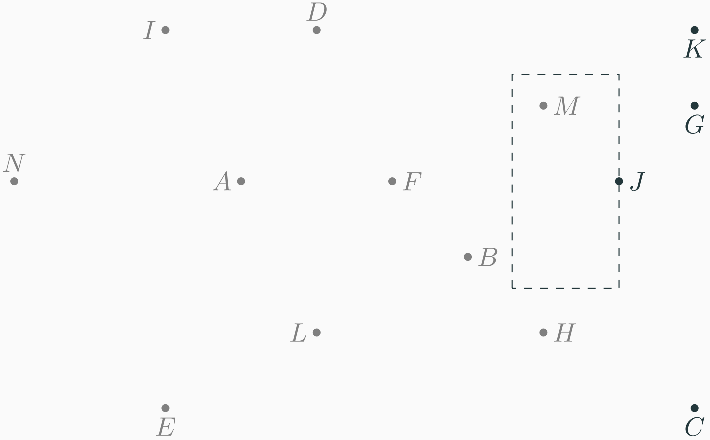
Visualização de identificação do par de pontos mais próximo

Avaliação do ponto M



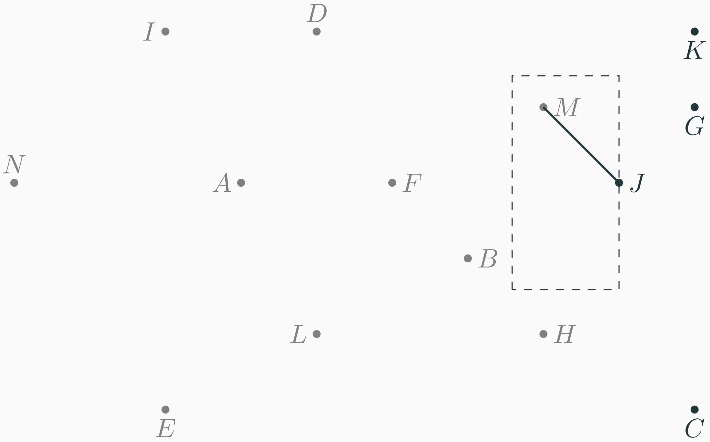
Visualização de identificação do par de pontos mais próximo

Avaliação do ponto J



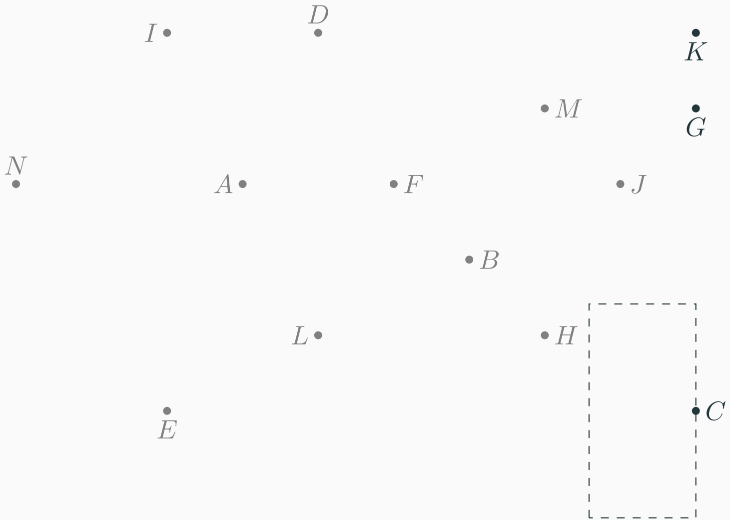
Visualização de identificação do par de pontos mais próximo

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



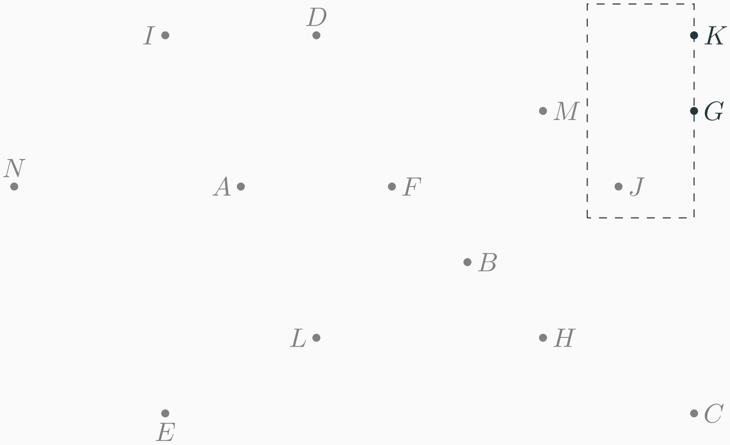
Visualização de identificação do par de pontos mais próximo

Avaliação do ponto C



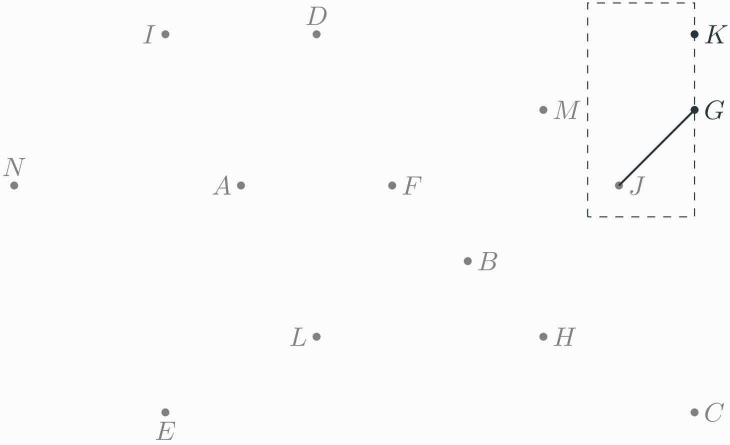
Visualização de identificação do par de pontos mais próximo

Avaliação do ponto G



Visualização de identificação do par de pontos mais próximo

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

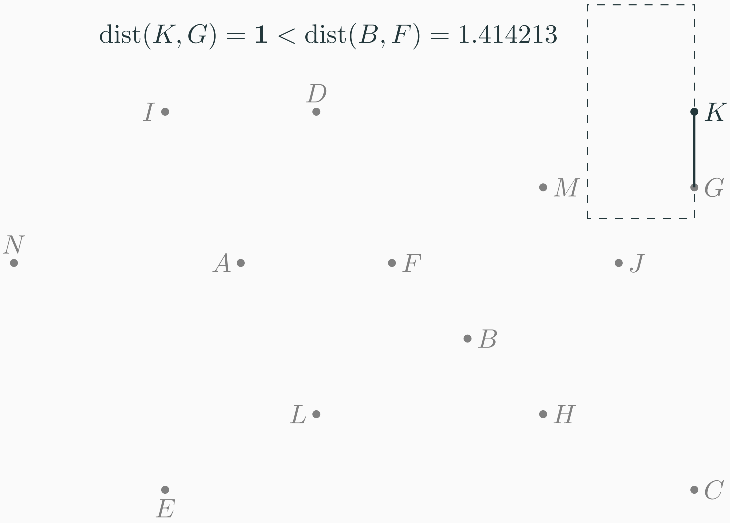


Visualização de identificação do par de pontos mais próximo

Avaliação do ponto K

Visualização de identificação do par de pontos mais próximo

$$\text{dist}(K, G) = \mathbf{1} < \text{dist}(B, F) = 1.414213$$



Implementação da identifacação do par mais próximo

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

using namespace std;
using ii = pair<double, double>;

struct Point { double x, y; };

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

pair<Point, Point> closest_pair(int N, vector<Point>& ps)
{
    sort(ps.begin(), ps.end());

    // Este código assume que N > 1
    auto d = dist(ps[0], ps[1]);
    auto closest = make_pair(ps[0], ps[1]);
```

Implementação da identifacação do par mais próximo

```
    set<ii> S;
    S.insert(ii(ps[0].y, ps[0].x));
    S.insert(ii(ps[1].y, ps[1].x));

    for (int i = 2; i < N; ++i)
    {
        auto P = ps[i];
        auto it = S.lower_bound(Point(P.y - d, 0));

        while (it != S.end())
        {
            auto Q = Point(it->second, it->first);

            if (Q.x < P.x - d)
            {
                it = S.erase(it);
                continue;
            }
```

Implementação da identifacação do par mais próximo

```
            if (Q.y > P.y + d)
                break;

            auto t = dist(P, Q);

            if (t < d)
            {
                d = t;
                closest = make_pair(P, Q);
            }

            ++it;
        }

        S.insert(ii(P.y, P.x));
    }

    return closest;
}
```

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

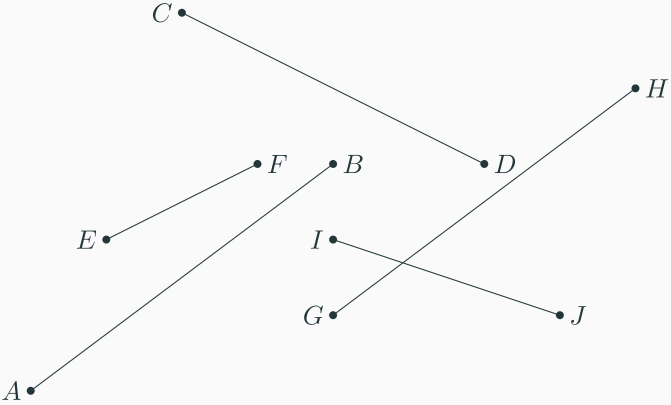
Algoritmo de Shamos-Hoey

- 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

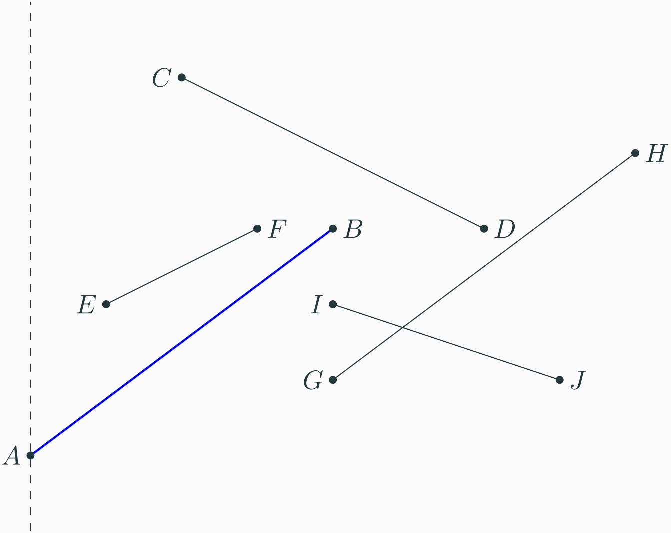
Algoritmo de Shamos-Hoey

- 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

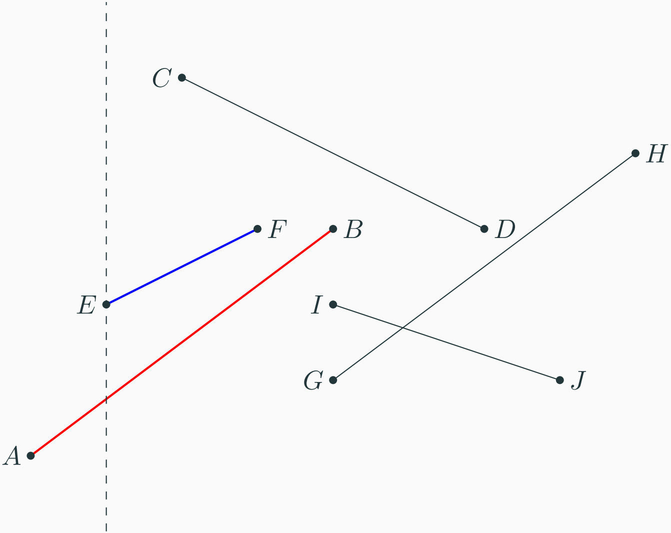
Visualização de identificação de interseção de segmentos



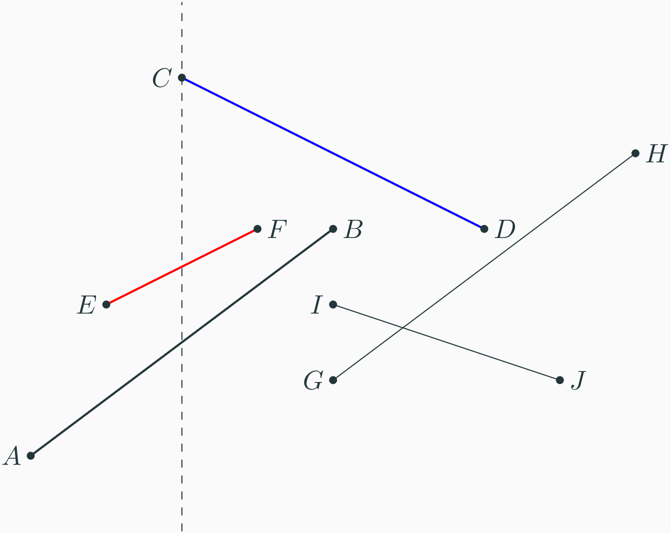
Visualização de identificação de interseção de segmentos



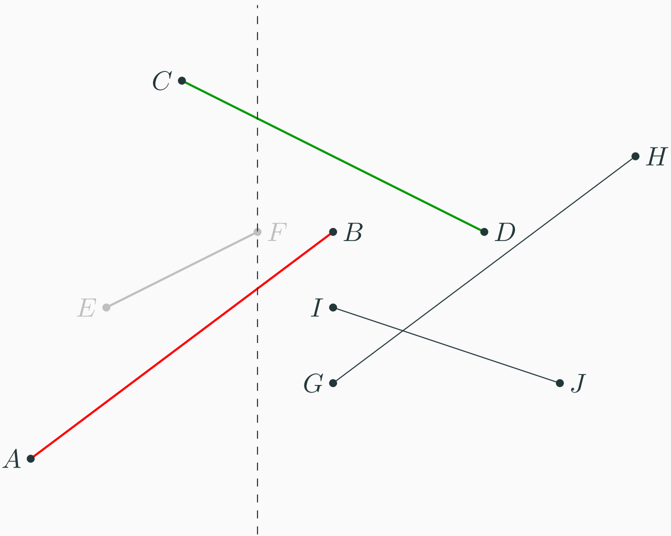
Visualização de identificação de interseção de segmentos



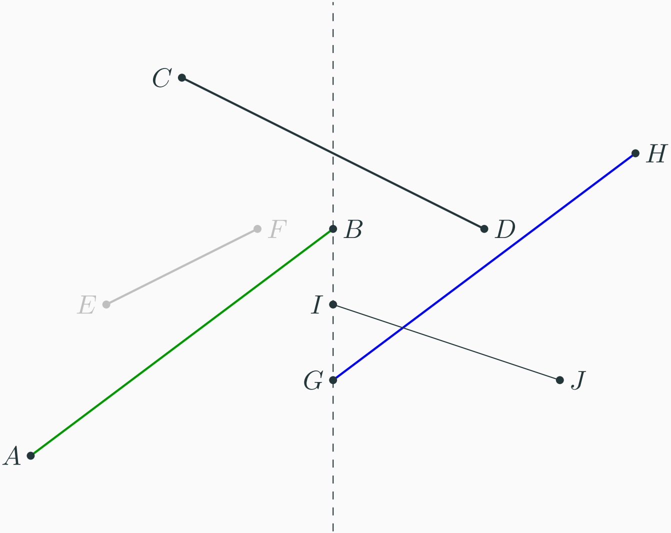
Visualização de identificação de interseção de segmentos



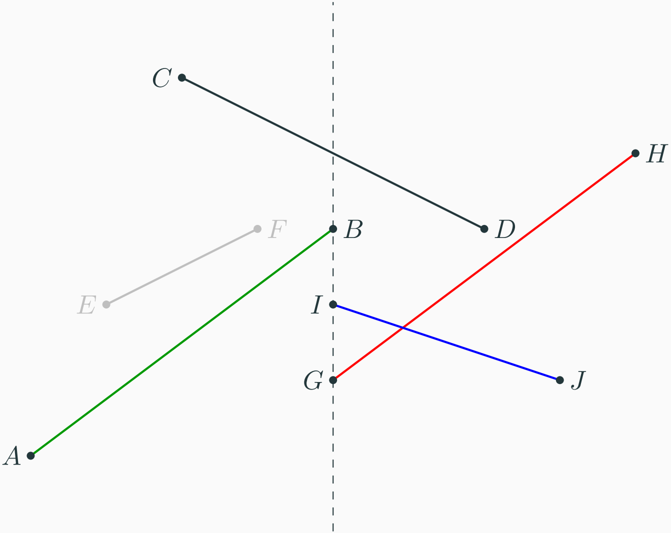
Visualização de identificação de interseção de segmentos



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

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

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

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

template<typename T>
T D(const Point<T>& P, const Point<T>& Q, const Point<T>& R)
{
    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);
}
```

Implementação do algoritmo de Shamos-Hoey

```cpp
template<typename T>
struct Segment
{
    T a, b, c;
    Point<T> A, B;

    Segment(const Point<T>& P, const Point<T>& Q)
        : 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; }

    bool operator<(const Segment& line) const
    {
        return (-a*sweep_x - c)*line.b < (-line.a*sweep_x -line.c)*b;
    }

    bool intersect(const Segment& s) const
    {
        auto d1 = D(A, B, s.A), d2 = D(A, B, s.B);

        if ((equals(d1, 0LL) && contains(s.A)) || (equals(d2, 0LL) && contains(s.B)))
            return true;
```

Implementação do algoritmo de Shamos-Hoey

```
        auto d3 = D(s.A, s.B, A);
        auto d4 = D(s.A, s.B, B);

        if ((equals(d3, 0LL) && s.contains(A)) || (equals(d4, 0LL) && s.contains(B)))
            return true;

        return (d1 * d2 < 0) && (d3 * d4 < 0);
    }

    bool contains(const Point<T>& P) const
    {
        if (P == A || P == B)
            return true;

        auto xmin = min(A.x, B.x);
        auto xmax = max(A.x, B.x);
        auto ymin = min(A.y, B.y);
        auto ymax = max(A.y, B.y);
```

Implementação do algoritmo de Shamos-Hoey

```
        if (P.x < xmin || P.x > xmax || P.y < ymin || P.y > ymax)
            return false;

        return equals((P.y - A.y)*(B.x - A.x), (P.x - A.x)*(B.y - A.y));
    }

    static T sweep_x;
};

template<typename T>
T Segment<T>::sweep_x;

template<typename T>
bool shamos_hoey(const vector&lt;Segment&lt;T&gt;&gt;& segments)
{
    struct Event
    {
        Point<T> P;
        size_t i;
```

Implementação do algoritmo de Shamos-Hoey

```
        bool operator<(const Event& e) const { return P < e.P; }
    };

    vector<Event> events;

    for (size_t i = 0; i < segments.size(); ++i)
    {
        events.push_back({ segments[i].A, i });
        events.push_back({ segments[i].B, i });
    }

    sort(events.begin(), events.end());
    set&lt;Segment<T>> sl;

    for (const auto& e : events)
    {
        auto s = segments[e.i];
        Segment<T>::sweep_x = e.P.x;
```

Implementação do algoritmo de Shamos-Hoey

```
        if (e.P == s.A)
        {
            sl.insert(s);

            auto it = sl.find(s);

            if (it != sl.begin())
            {
                auto L = *prev(it);

                if (s.intersect(L)) return true;
            }

            if (next(it) != sl.end())
            {
                auto U = *next(it);

                if (s.intersect(U)) return true;
            }
```

Implementação do algoritmo de Shamos-Hoey

```
        } else
        {
            auto it = sl.find(s);

            if (it != sl.begin() and it != sl.end())
            {
                auto L = *prev(it);
                auto U = *next(it);

                if (L.intersect(U)) return true;
            }

            sl.erase(it);
        }
    }

    return false;
}
```

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

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

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

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

template<typename T>
struct Segment
{
    T a, b, c;
    Point<T> A, B;
```

Implementação do algoritmo de busca completa

```cpp
    Segment(const Point<T>& P, const Point<T>& Q)
        : a(P.y - Q.y), b (Q.x - P.x), c(P.x*Q.y - Q.x*P.y), A(P), B(Q) { }

    optional<Point<T>> intersection(const Segment& s)
    {
        auto det = a * s.b - b * s.a;

        if (not equals(det, 0.0))     // Concorrentes
        {
            auto x = (-c * s.b + s.c * b) / det;
            auto y = (-s.c * a + c * s.a) / det;

            if (min(A.x, B.x) <= x and x <= max(A.x, B.x) and
                min(s.A.x, s.B.x) <= x and x <= max(s.A.x, s.B.x))
            {
                return Point<T> { x, y };
            }
        }

        return { };
```

Implementação do algoritmo de busca completa

```cpp
template<typename T>
set<Point<T>> intersections(int N, const vector&lt;Segment&lt;T&gt;&gt;&amp; segments)
{
    set<Point<T>> ans;

    for (int i = 0; i < N; ++i)
    {
        auto s = segments[i];

        for (int j = i + 1; j < N; ++j)
        {
            auto r = segments[j];
            auto P = s.intersection(r);

            if (P) ans.insert(P.value());
        }
    }

    return ans;
}
```

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

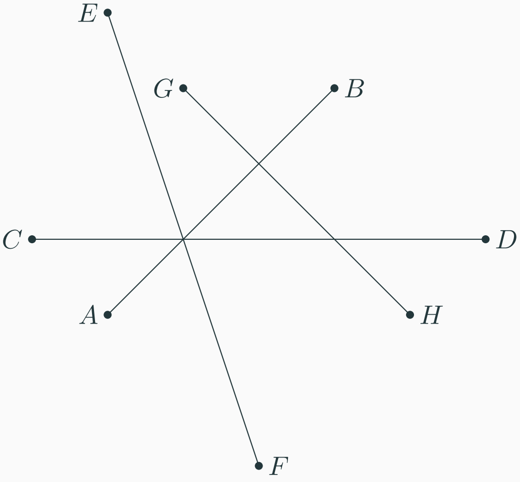
Algoritmo de Bentley-Ottman

- 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

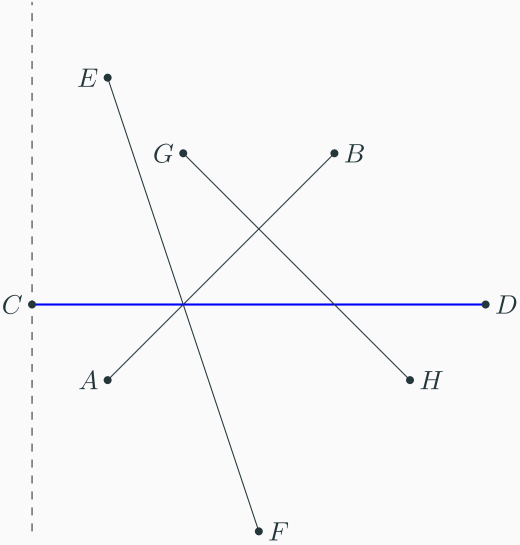
Algoritmo de Bentley-Ottman

- 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

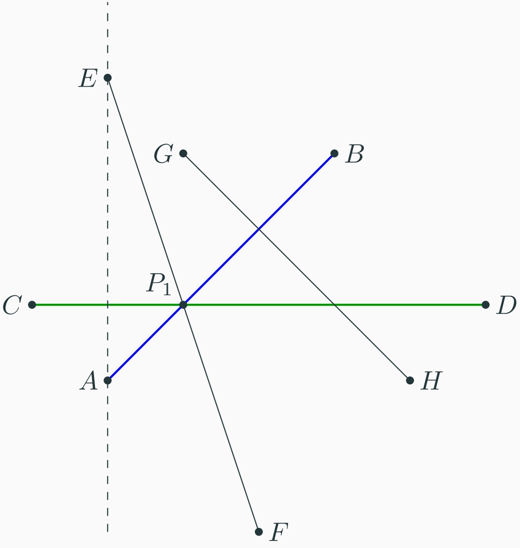
Visualização da contagem das interseções entres segmentos



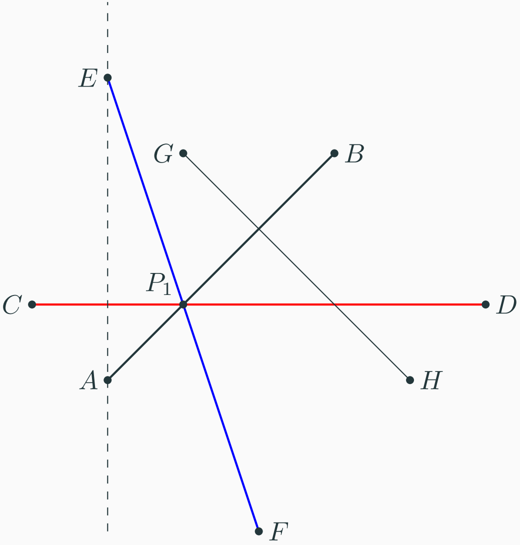
Visualização da contagem das interseções entres segmentos



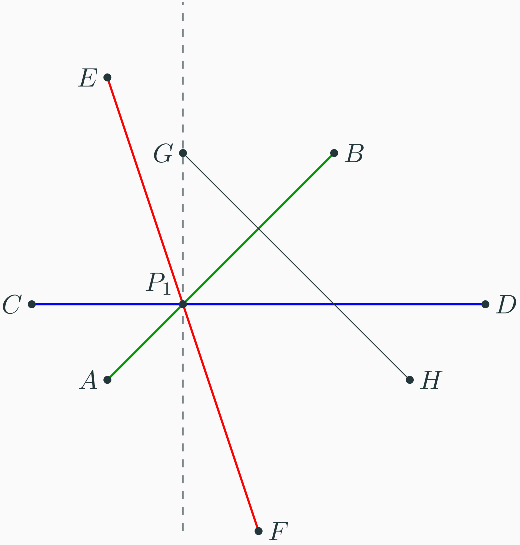
Visualização da contagem das interseções entres segmentos



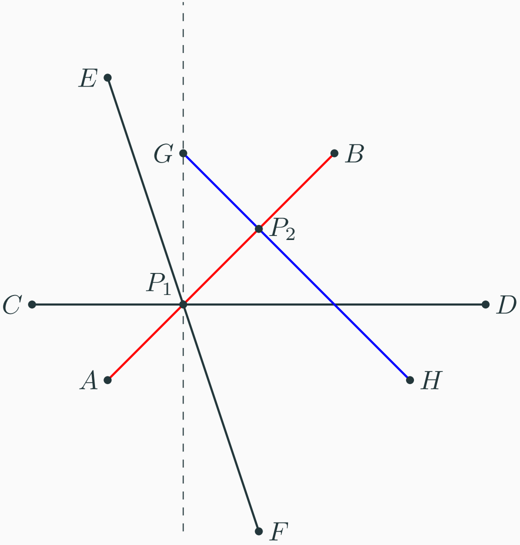
Visualização da contagem das interseções entres segmentos



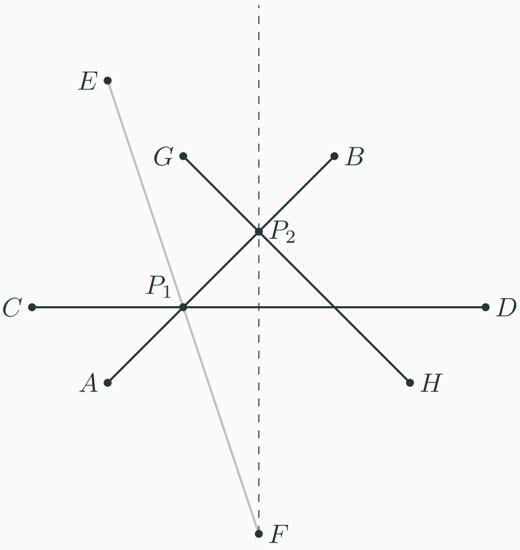
Visualização da contagem das interseções entres segmentos



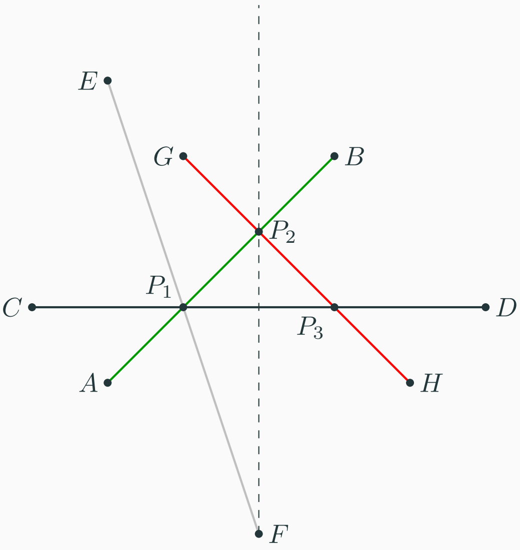
Visualização da contagem das interseções entres segmentos



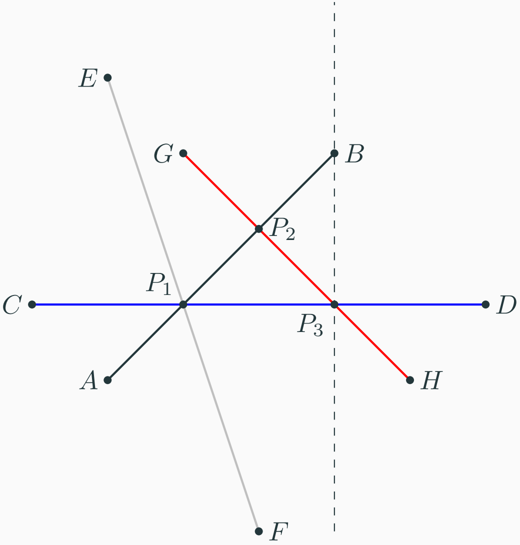
Visualização da contagem das interseções entres segmentos



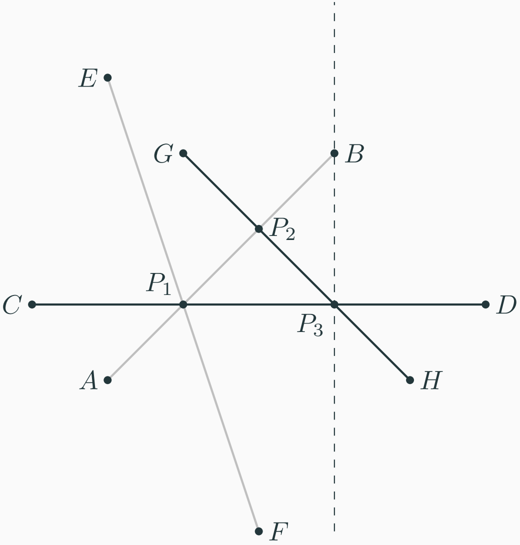
Visualização da contagem das interseções entres segmentos



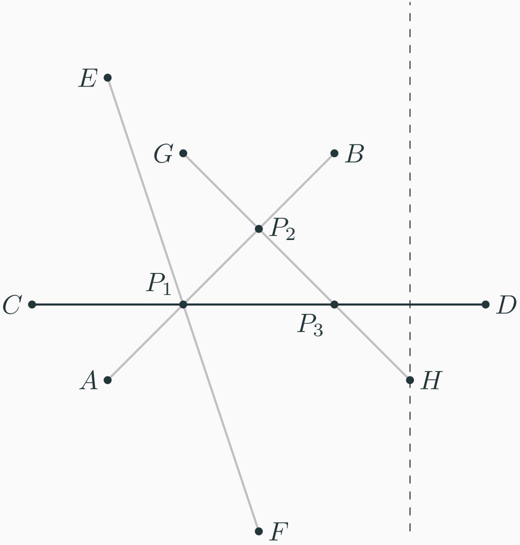
Visualização da contagem das interseções entres segmentos



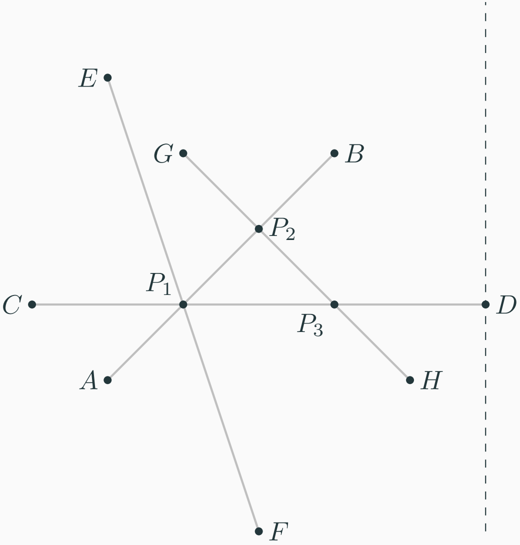
Visualização da contagem das interseções entres segmentos



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;
    }
```

```
    bool operator==(const Point& P) const
    {
        return x == P.x and y == P.y;
    }

    bool operator!=(const Point& P) const
    {
        return not (*this == P);
    }
};

struct Segment
{
    double a, b, c;
    Point A, B;
    size_t idx;

    Segment(const Point& P, const Point& Q, size_t i)
        : 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

```cpp
    bool operator<(const Segment& s) const
    {
        return (-a*sweep_x - c)*s.b < (-s.a*sweep_x -s.c)*b;
    }

    optional<Point> intersection(const Segment& s) const
    {
        auto det = a * s.b - b * s.a;

        if (not equals(det, 0.0))     // Concorrentes
        {
            auto x = (-c * s.b + s.c * b) / det;
            auto y = (-s.c * a + c * s.a) / det;

            if (min(A.x, B.x) <= x and x <= max(A.x, B.x) and
                min(s.A.x, s.B.x) <= x and x <= max(s.A.x, s.B.x))
            {
                return Point { x, y };
            }
        }
```

Implementação do algoritmo de Bentley-Ottman

```cpp
            return { };
        }

        static double sweep_x;
};

double Segment::sweep_x;

struct Event
{
    enum Type { OPEN, INTERSECTION, CLOSE };

    Point P;
    Type type;
    size_t i;

    bool operator<(const Event& e) const
    {
        if (P != e.P)
            return e.P < P;
```

Implementação do algoritmo de Bentley-Ottman

```
        if (type != e.type)
            return type > e.type;

        return i > e.i;
    }
};

void add_neighbor_intersections(const Segment& s, const set&lt;Segment&gt;& sl,
    set<Point>& ans, priority_queue<Event>& events)
{
    // TODO: garantir que a busca identifique unicamente o elemento s,
    // através do ajuste fino da variável Segment::sweep_x
    auto it = sl.find(s);

    if (it != sl.begin())
    {
        auto L = *prev(it);
        auto P = s.intersection(L);
```

Implementação do algoritmo de Bentley-Ottman

```
        if (P and ans.count(P.value()) == 0)
        {
            events.push(Event { P.value(), Event::INTERSECTION, s.idx } );
            ans.insert(P.value());
        }
    }

    if (next(it) != sl.end())
    {
        auto U = *next(it);
        auto P = s.intersection(U);

        if (P and ans.count(P.value()) == 0)
        {
            events.push(Event { P.value(), Event::INTERSECTION, s.idx } );
            ans.insert(P.value());
        }
    }
}
```

Implementação do algoritmo de Bentley-Ottman

```
set<Point> bentley_ottman(vector&lt;Segment&gt;& segments)
{
    set<Point> ans;
    priority_queue<Event> events;

    for (size_t i = 0; i < segments.size(); ++i)
    {
        events.push(Event { segments[i].A, Event::OPEN, i });
        events.push(Event { segments[i].B, Event::CLOSE, i });
    }

    set&lt;Segment&gt; sl;

    while (not events.empty())
    {
        auto e = events.top();
        events.pop();

        Segment::sweep_x = e.P.x;
```

Implementação do algoritmo de Bentley-Ottman

```
switch (e.type) {
case Event::OPEN:
{
    auto s = segments[e.i];
    sl.insert(s);

    add_neighbor_intersections(s, sl, ans, events);
}
break;

case Event::CLOSE:
{
    auto s = segments[e.i];
    auto it = sl.find(s);       // TODO: aqui também

    if (it != sl.begin() and it != sl.end())
    {
        auto L = *prev(it);
        auto U = *next(it);
        auto P = L.intersection(U);
```

Implementação do algoritmo de Bentley-Ottman

```
                if (P and ans.count(P.value()) == 0)
                    events.push( Event { P.value(), Event::INTERSECTION, L.idx } );
            }

            sl.erase(it);
        }
        break;

    default:
        auto r = segments[e.i];
        auto p = sl.equal_range(r);

        vector&lt;Segment&gt; range(p.first, p.second);

        // Remove os segmentos que se interceptam
        sl.erase(p.first, p.second);

        // Reinsere os segmentos
        Segment::sweep_x += 0.1;
```

Implementação do algoritmo de Bentley-Ottman

```cpp
            sl.insert(range.begin(), range.end());

            // Procura interseções com os novos vizinhos
            for (const auto& s : range)
                add_neighbor_intersections(s, sl, ans, events);
        }
    }

    return ans;
}
```

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.