Geometria Computacional

Sweep line: algoritmos

Prof. Edson Alves

2019

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

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

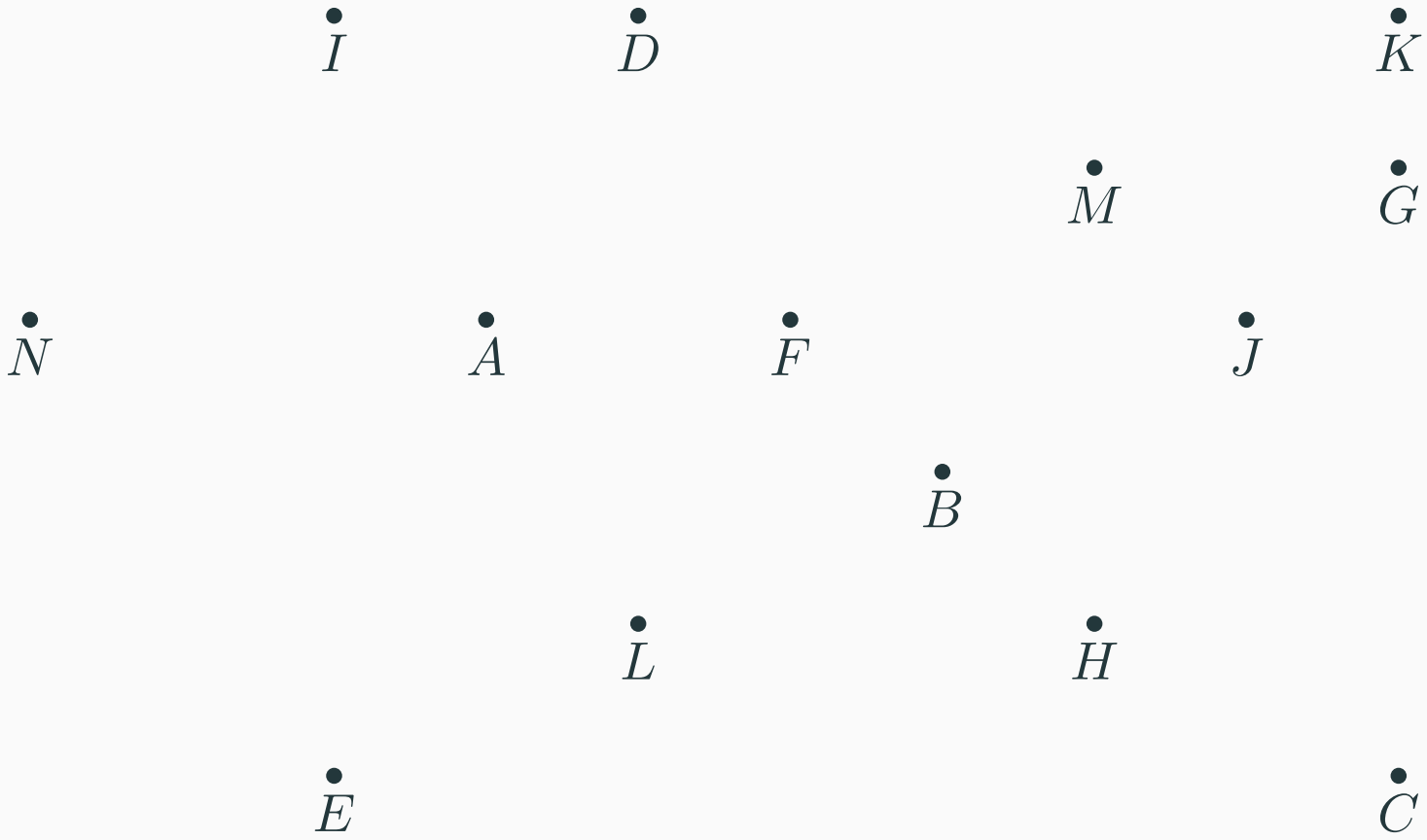
- Uma abordagem de busca completa consiste em computar as distância entre todos os pares de pontos possível, tendo complexidade $O(N^2)$
- Contudo, o problema pode ser resolvido em $O(N \log N)$ através do *sweep line*
- Os pontos deve ser ordenados em ordem lexicográfica

Par de pontos mais próximo

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

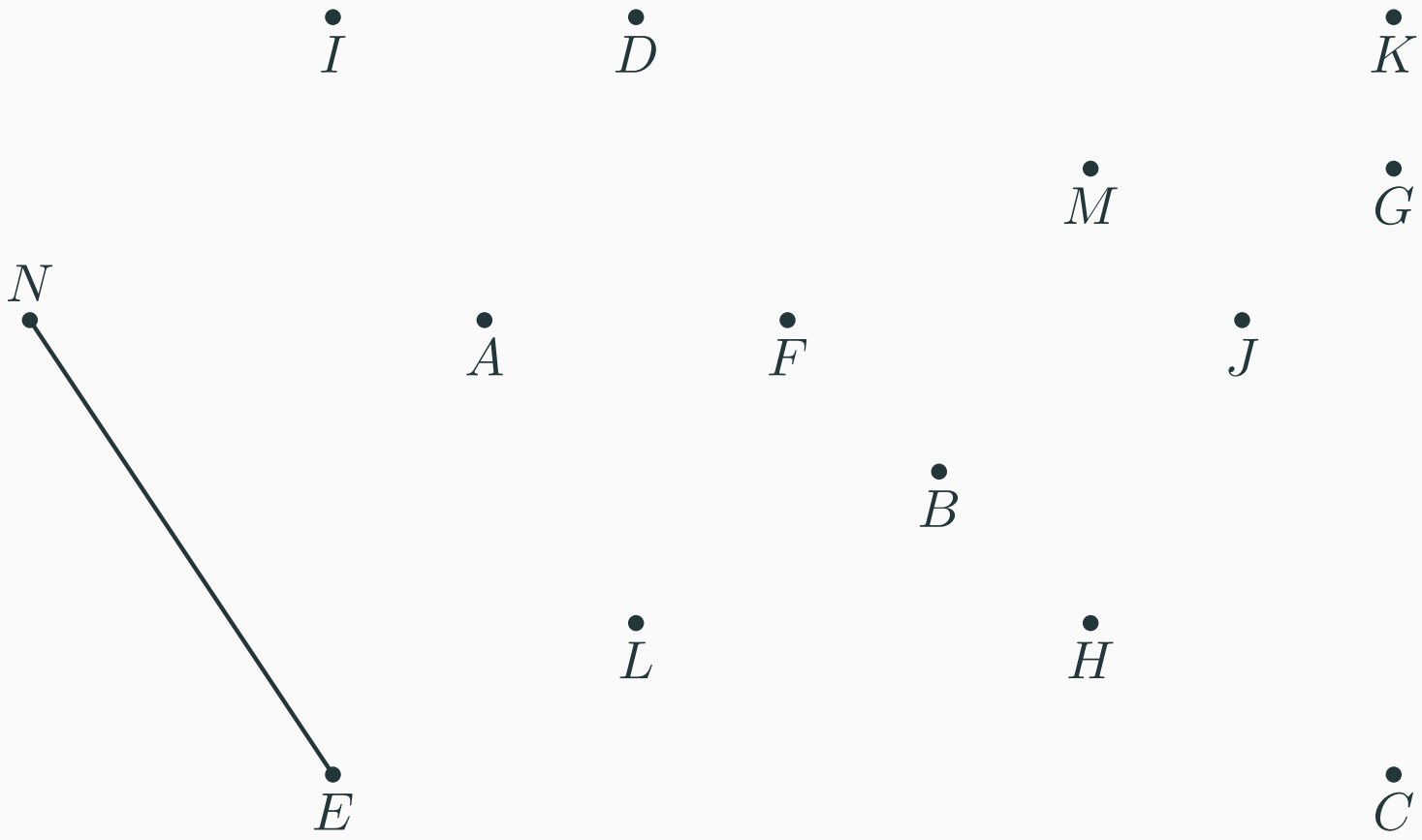
Visualização de identificação do par de pontos mais próximo

Par mais próximo: -



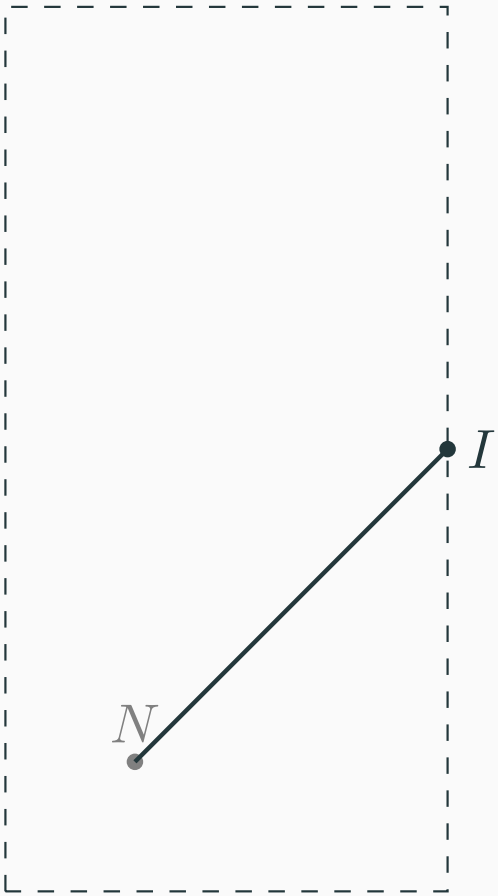
Visualização de identificação do par de pontos mais próximo

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

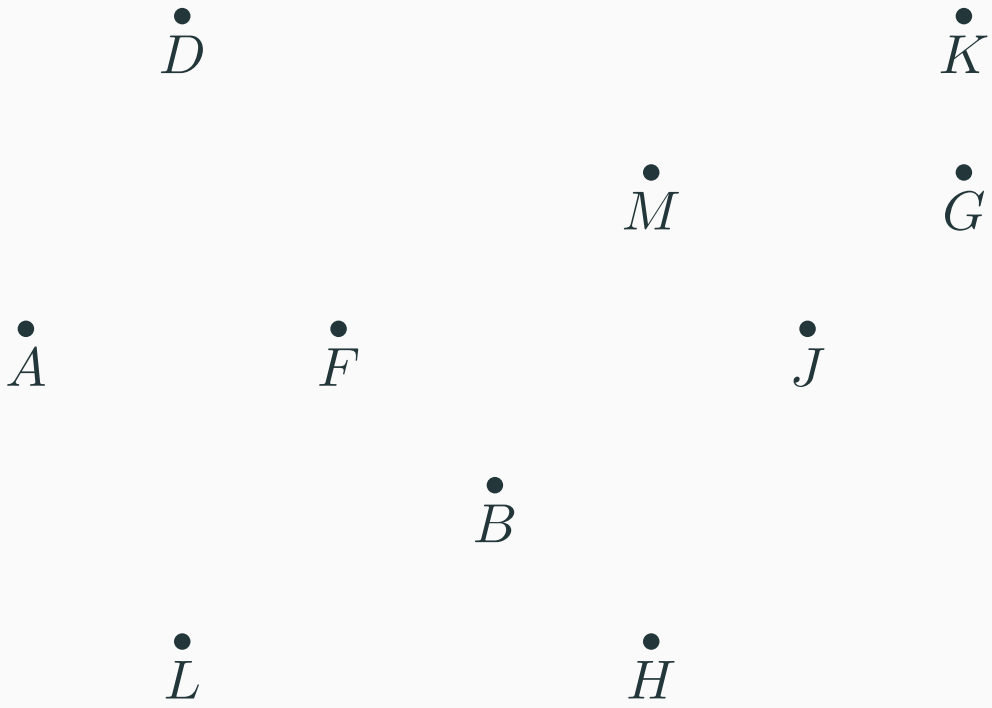


Visualização de identificação do par de pontos mais próximo

Visualização de identificação do par de pontos mais próximo



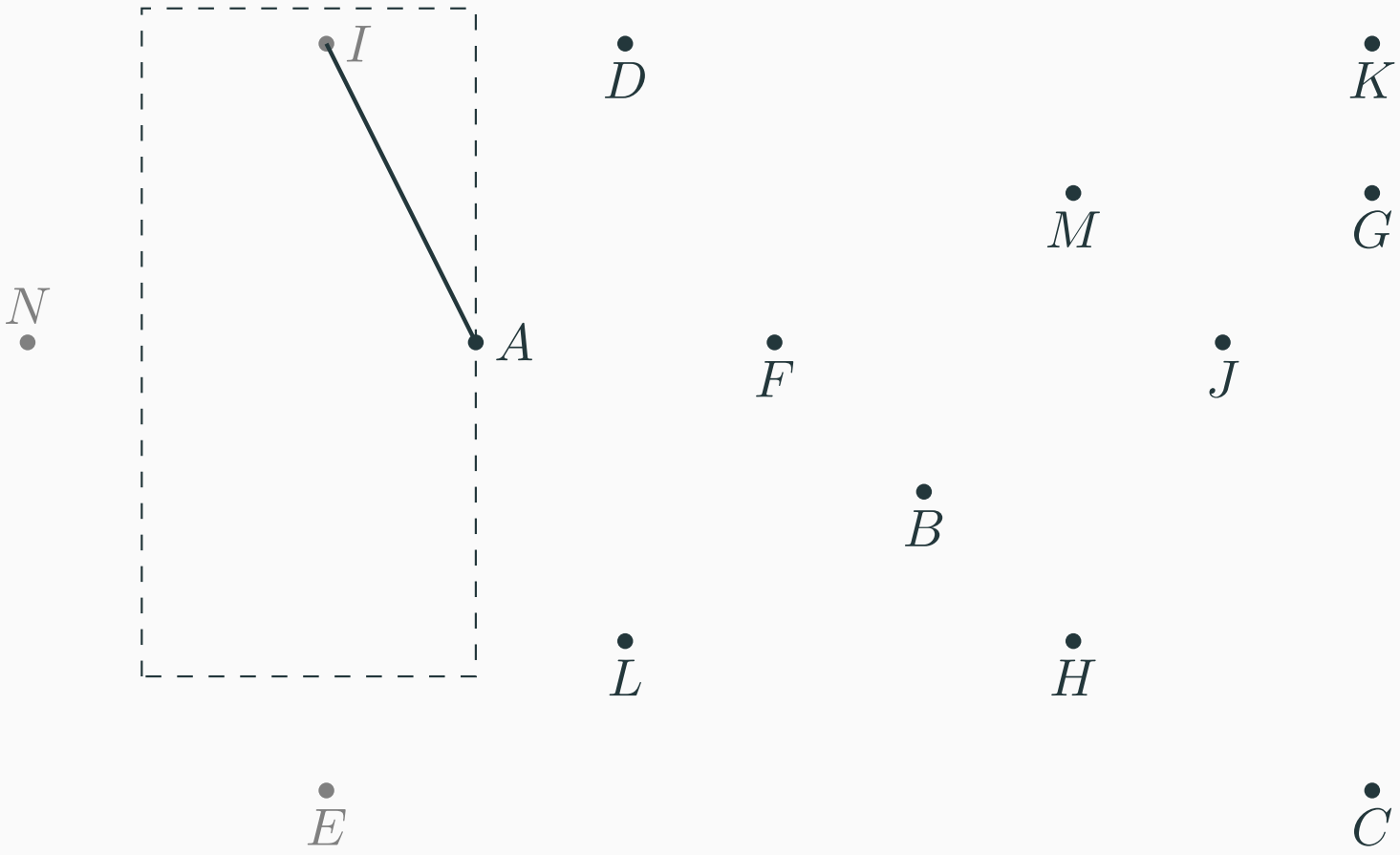
$$\text{dist}(I, N) = \mathbf{2.828427} < \text{dist}(N, E) = 3.605551$$



Visualização de identificação do par de pontos mais próximo

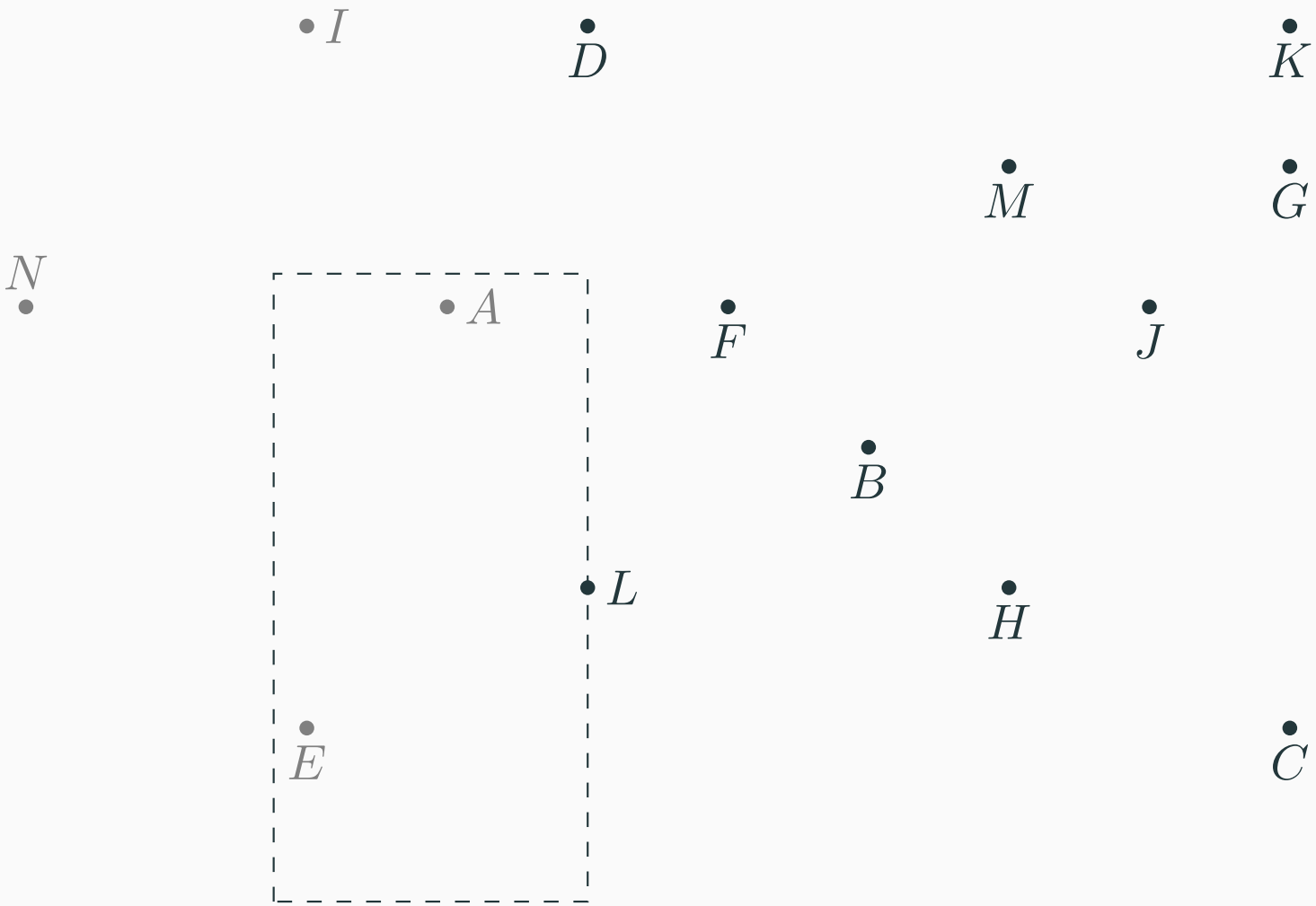
Visualização de identificação do par de pontos mais próximo

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



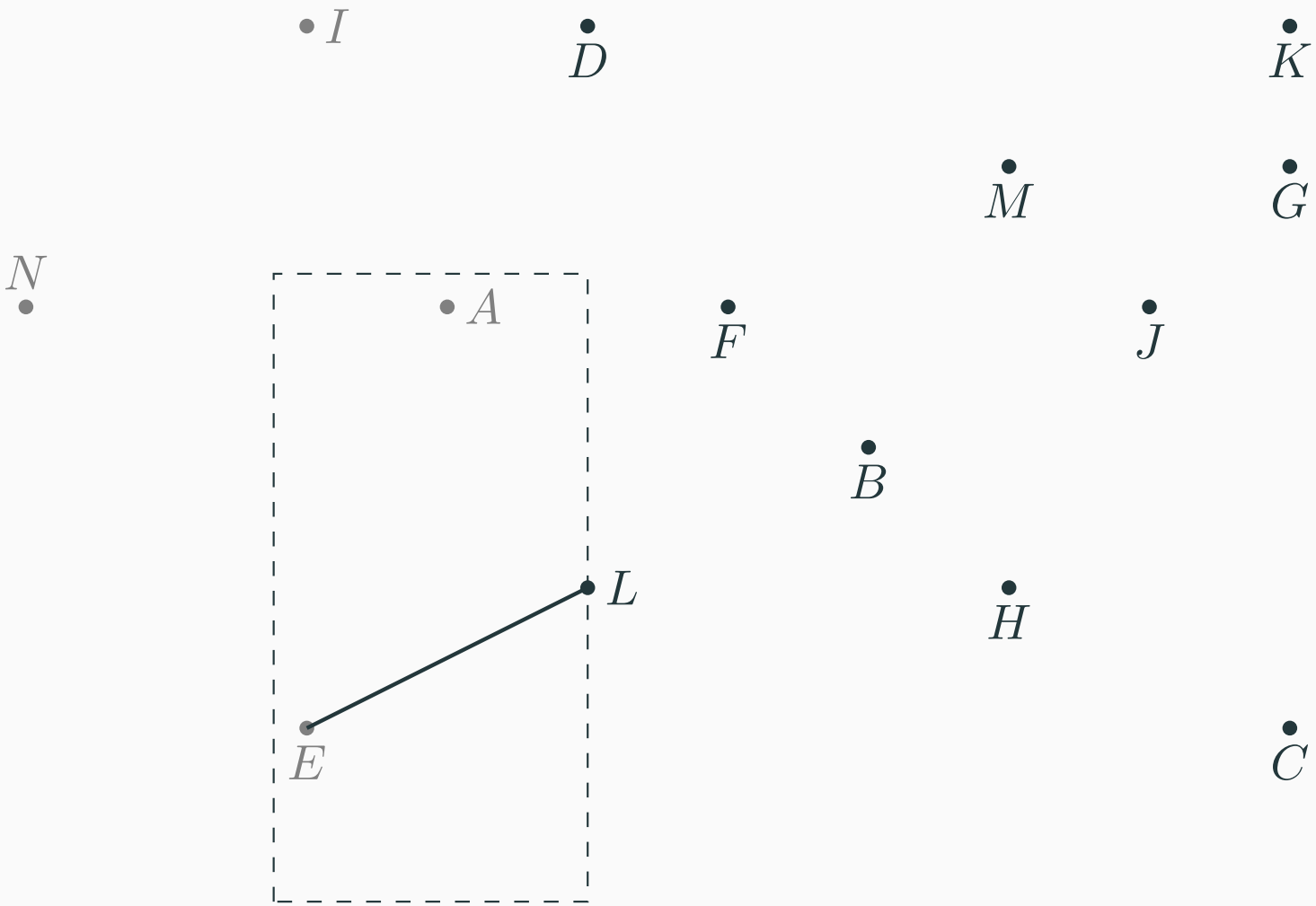
Visualização de identificação do par de pontos mais próximo

Avaliação do ponto L



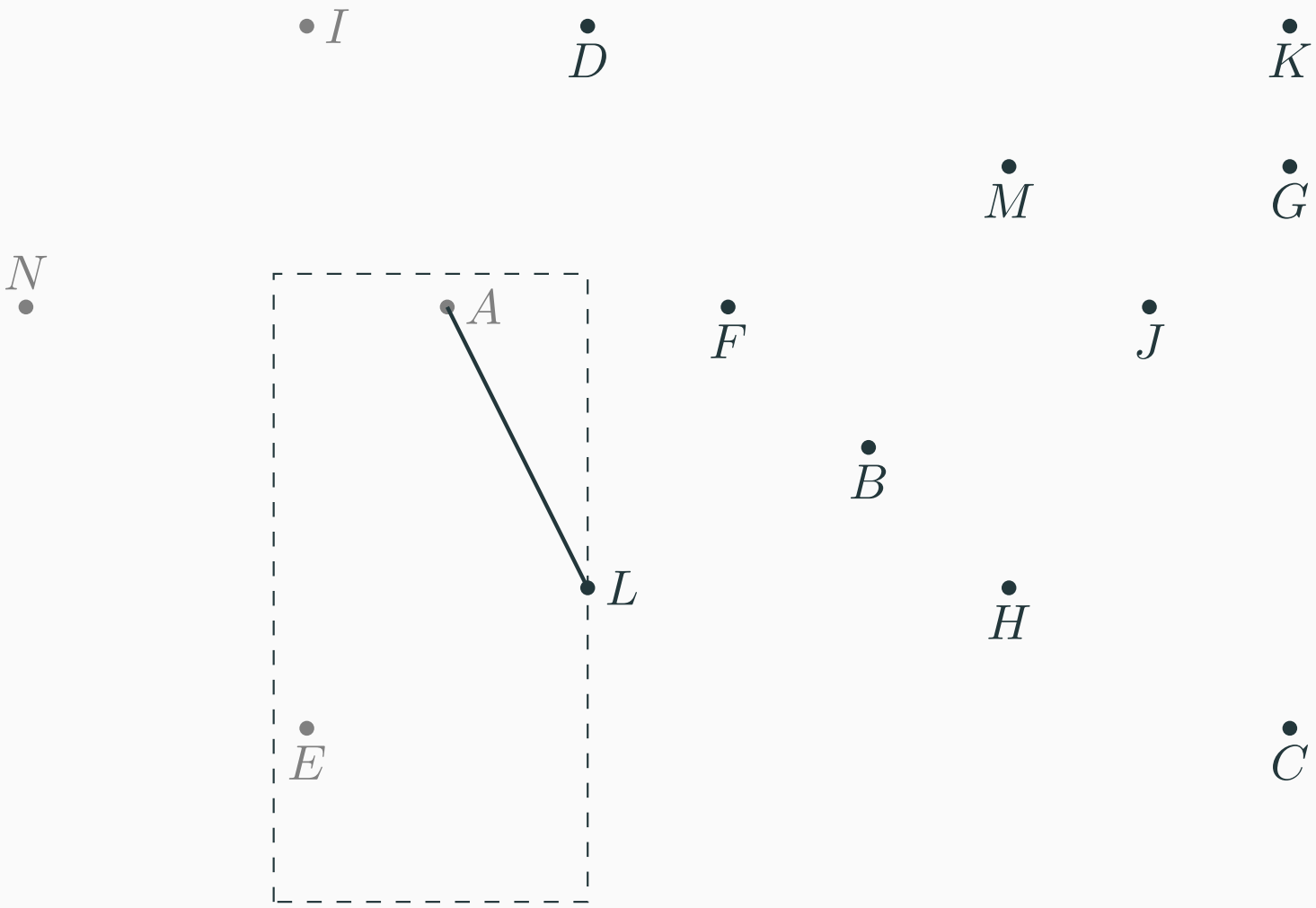
Visualização de identificação do par de pontos mais próximo

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



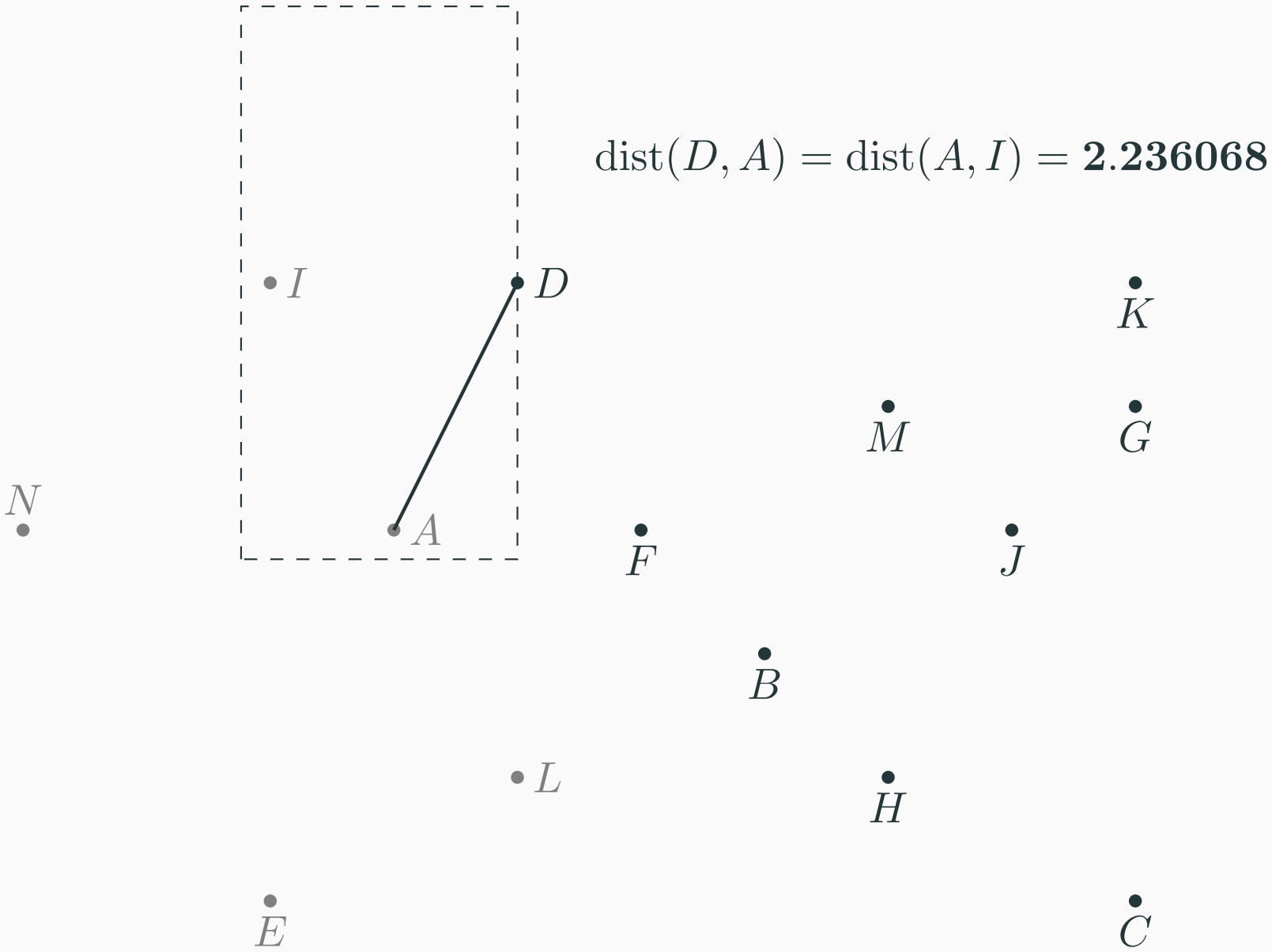
Visualização de identificação do par de pontos mais próximo

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

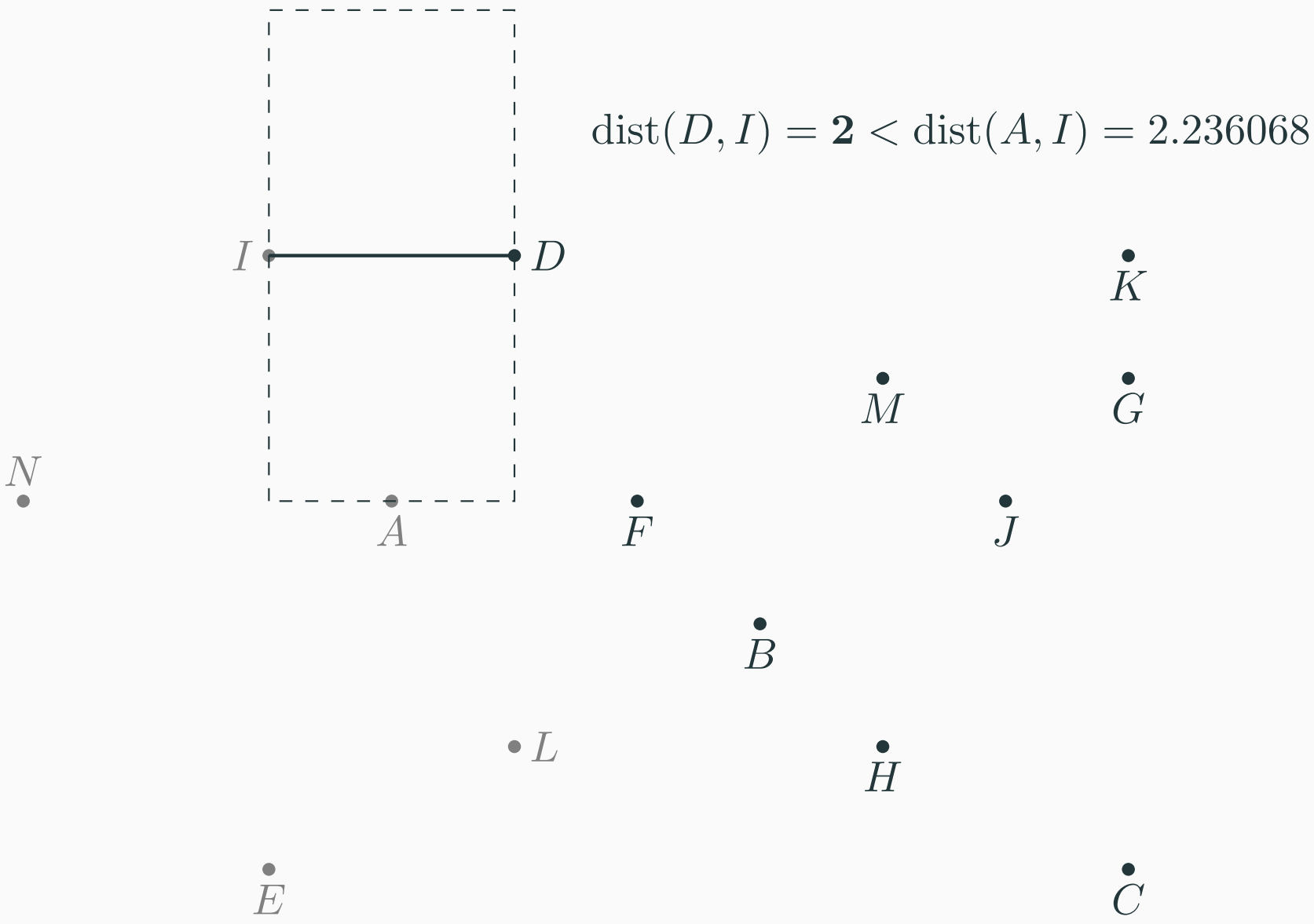


Visualização de identificação do par de pontos mais próximo

Visualização de identificação do par de pontos mais próximo



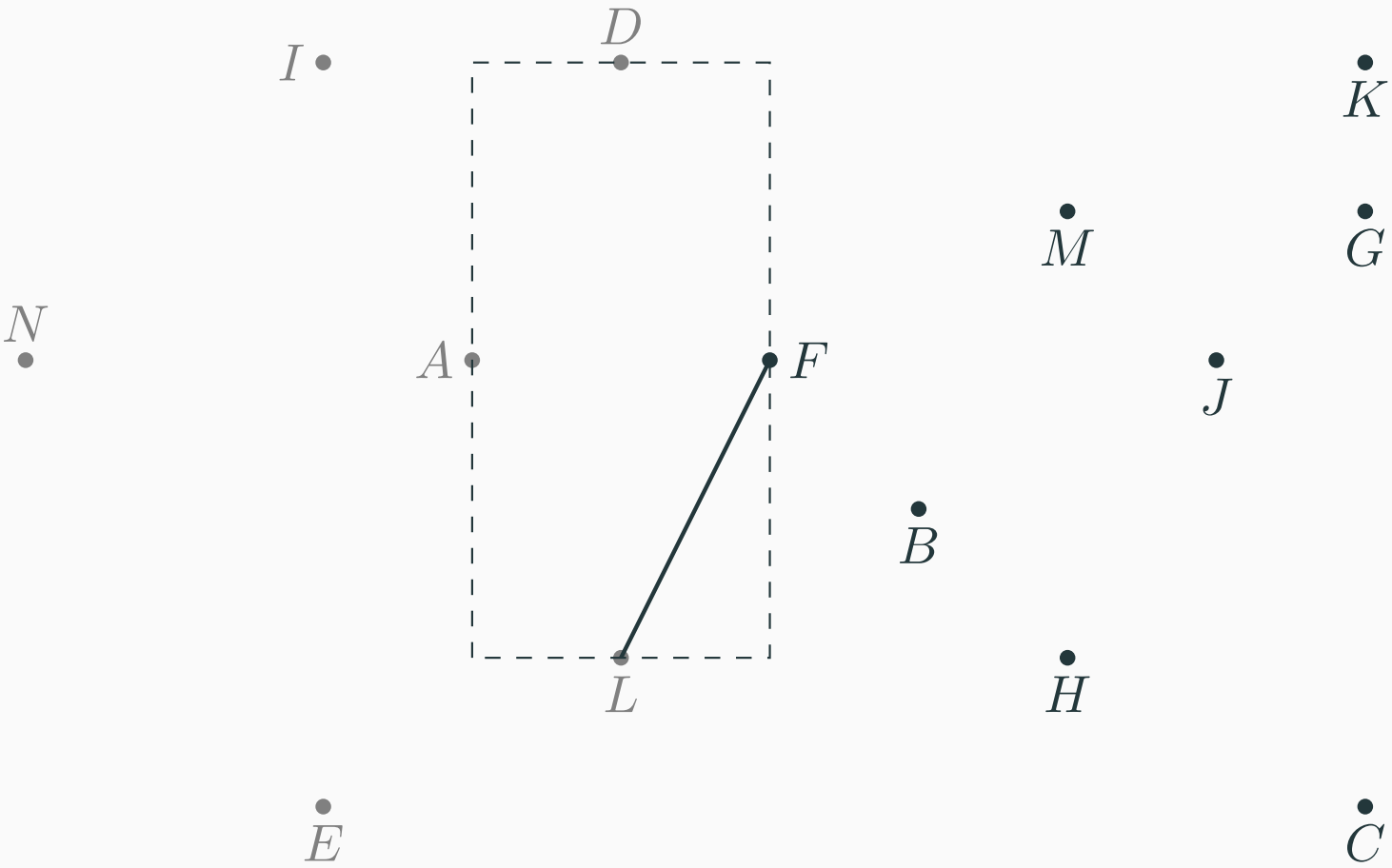
Visualização de identificação do par de pontos mais próximo



Visualização de identificação do par de pontos mais próximo

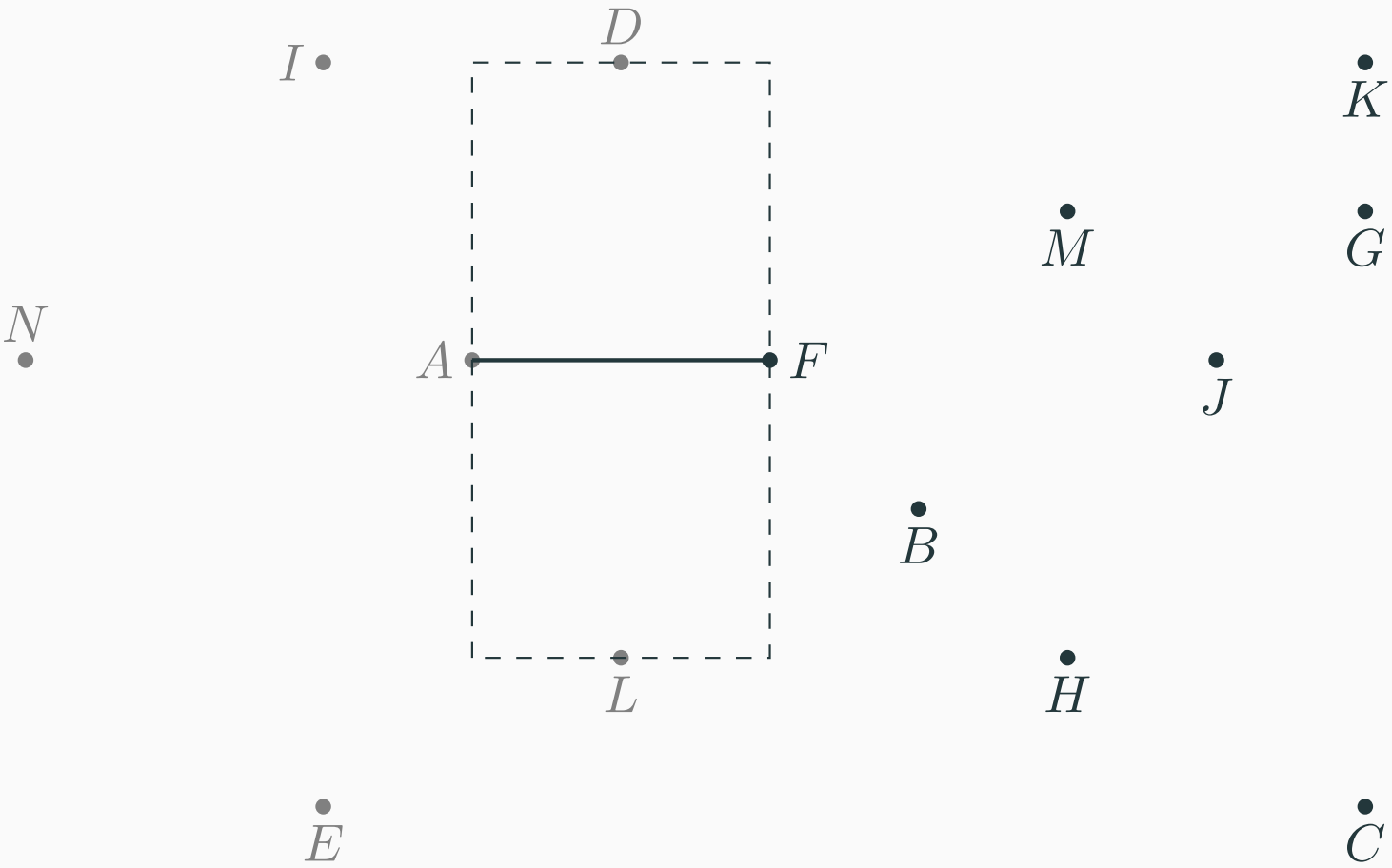
Visualização de identificação do par de pontos mais próximo

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



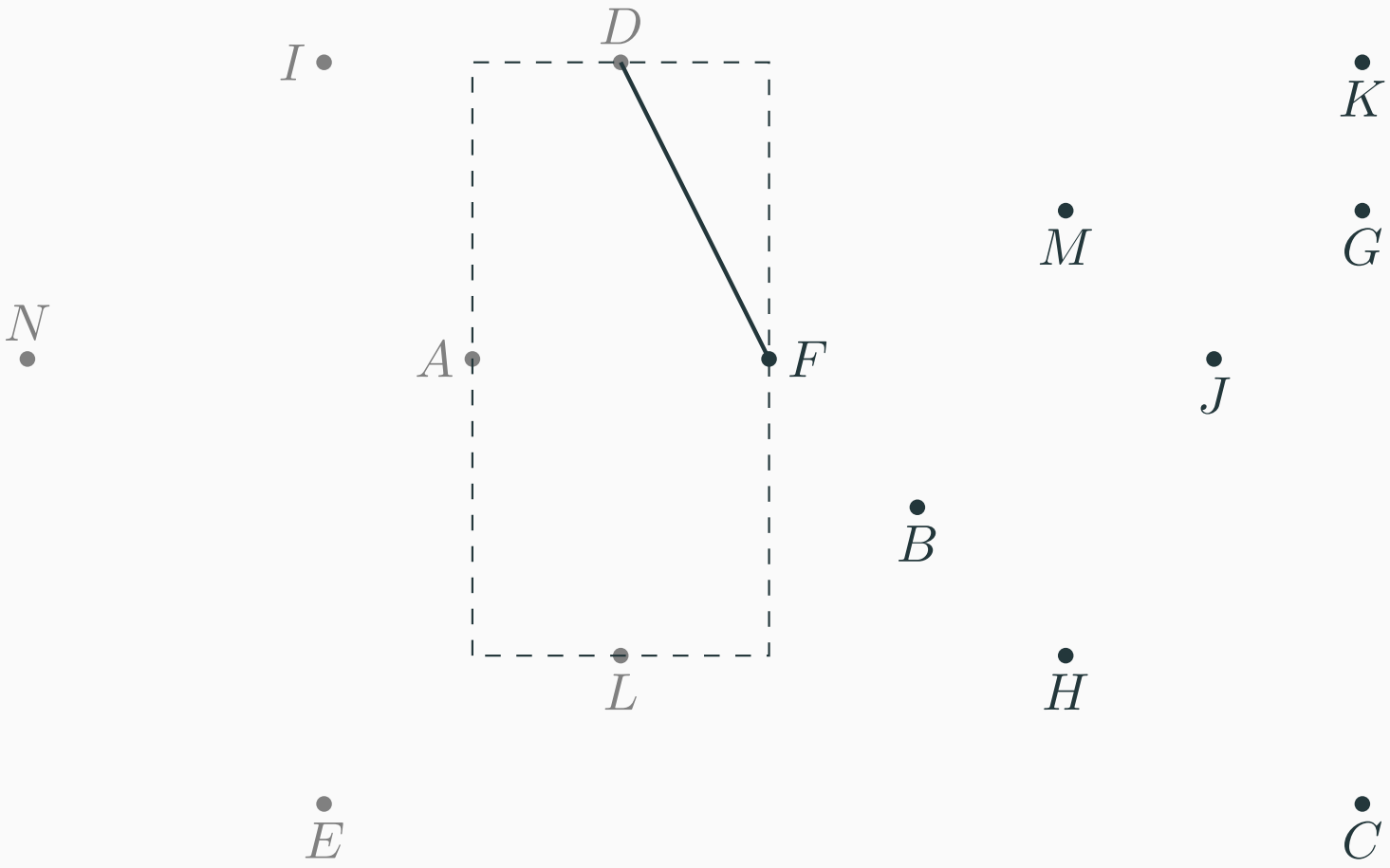
Visualização de identificação do par de pontos mais próximo

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

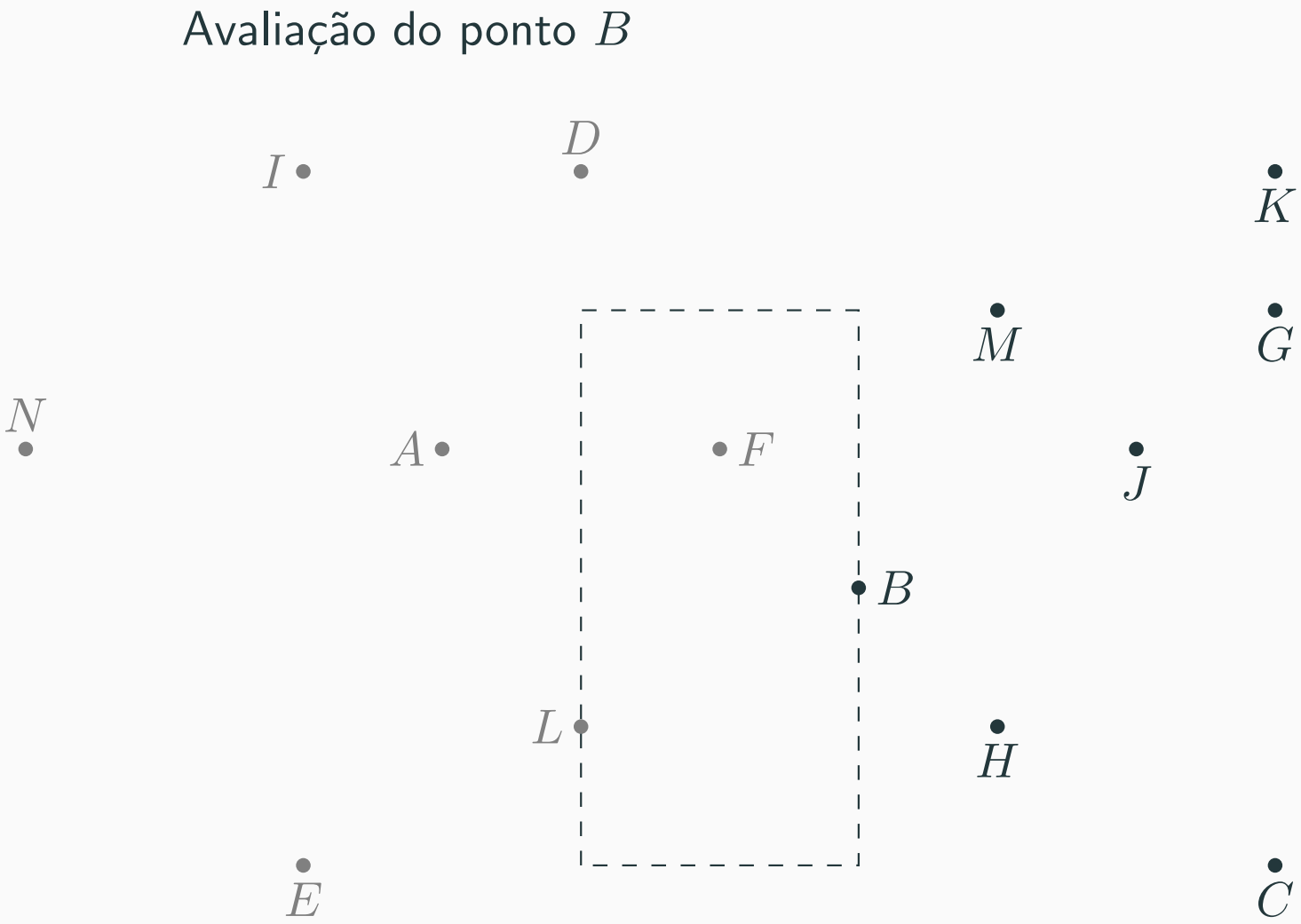


Visualização de identificação do par de pontos mais próximo

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

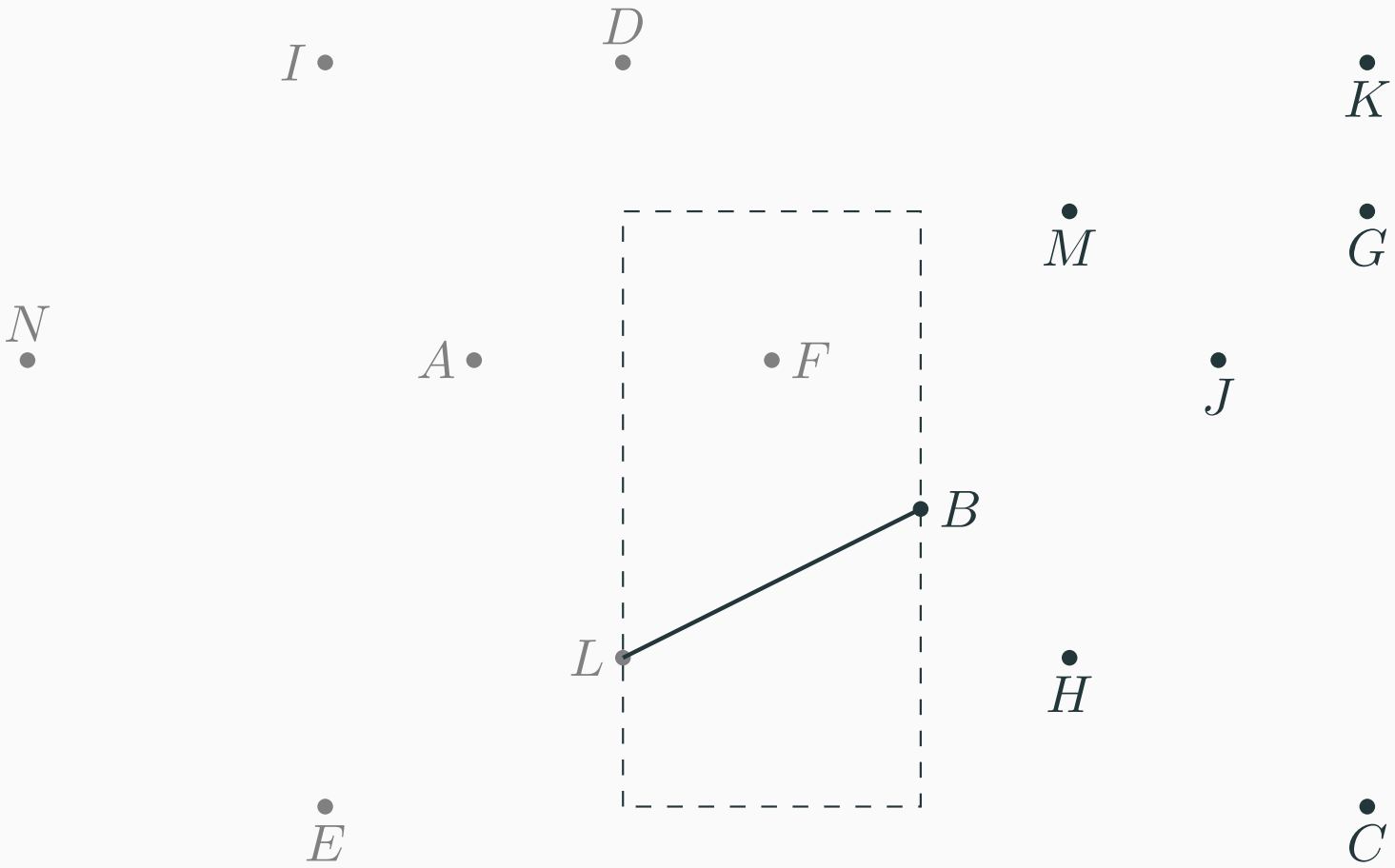


Visualização de identificação do par de pontos mais próximo



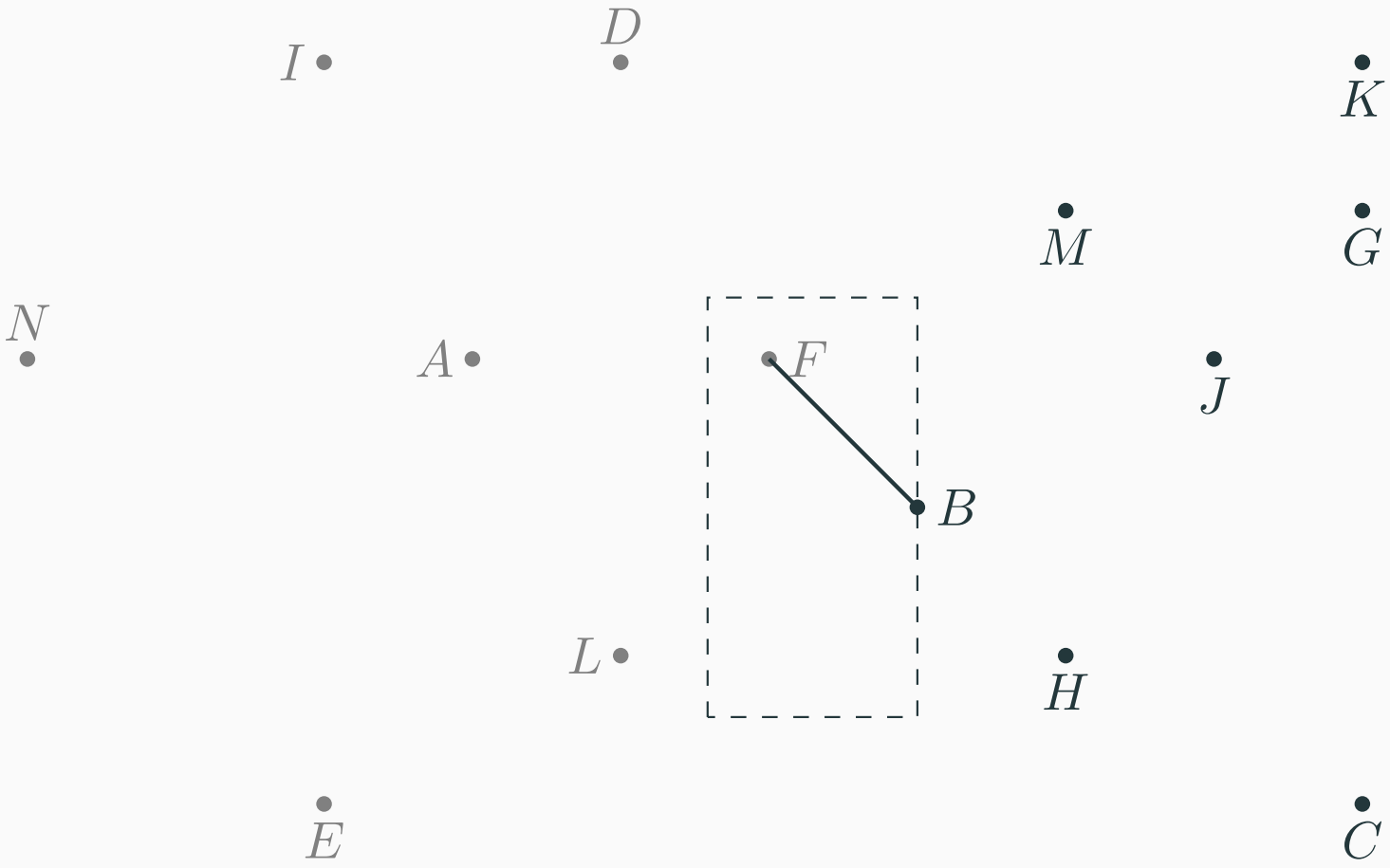
Visualização de identificação do par de pontos mais próximo

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

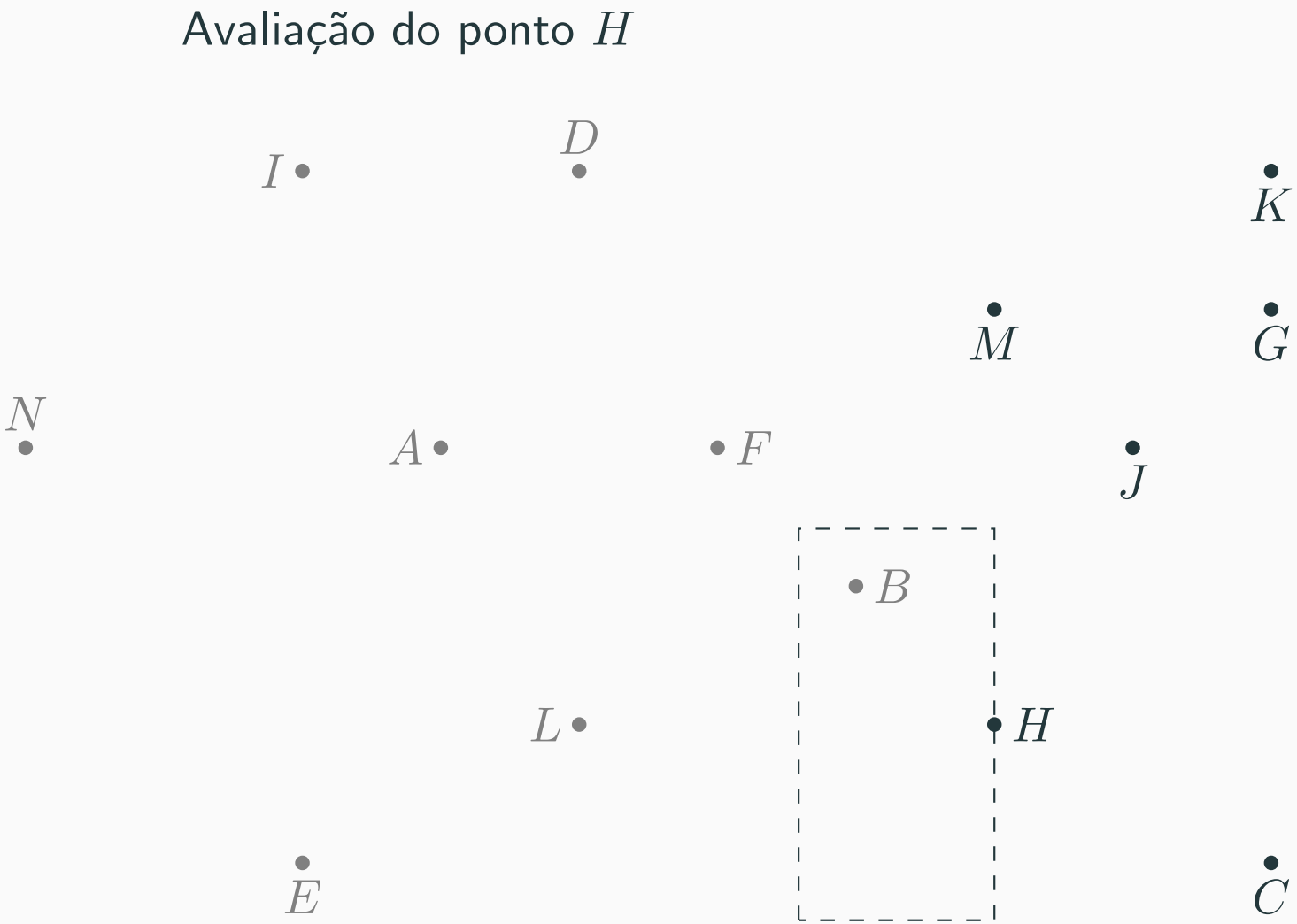


Visualização de identificação do par de pontos mais próximo

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

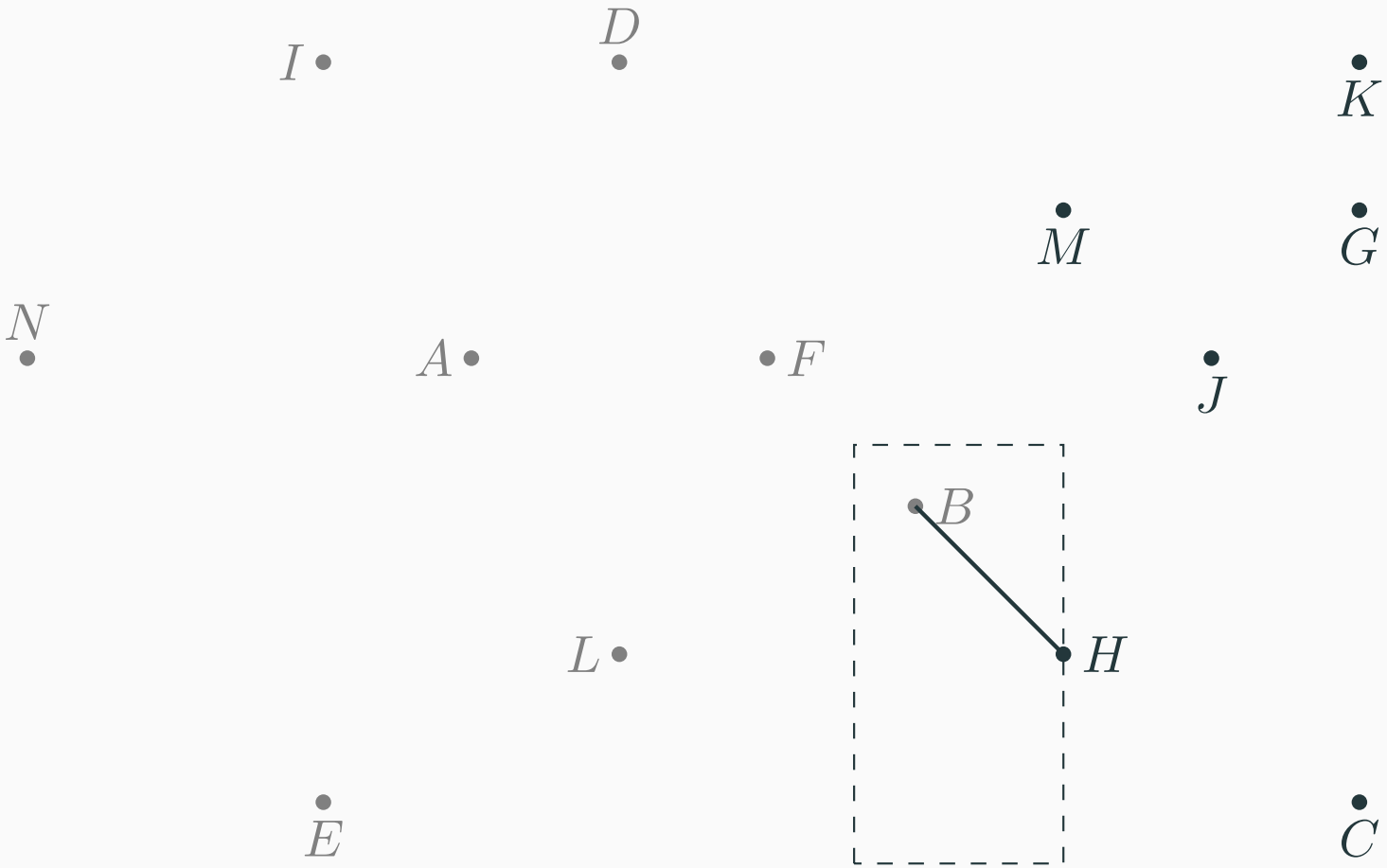


Visualização de identificação do par de pontos mais próximo

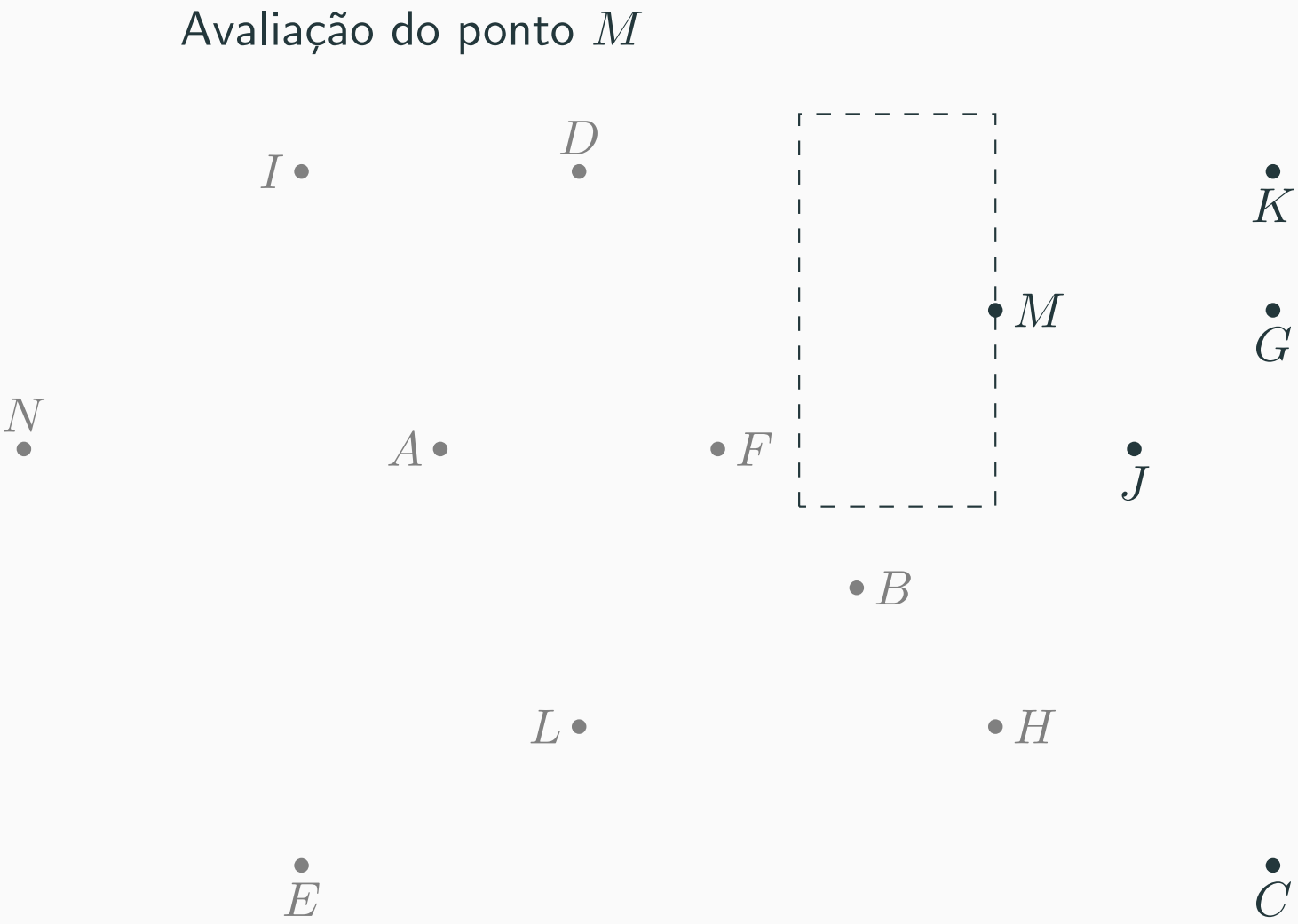


Visualização de identificação do par de pontos mais próximo

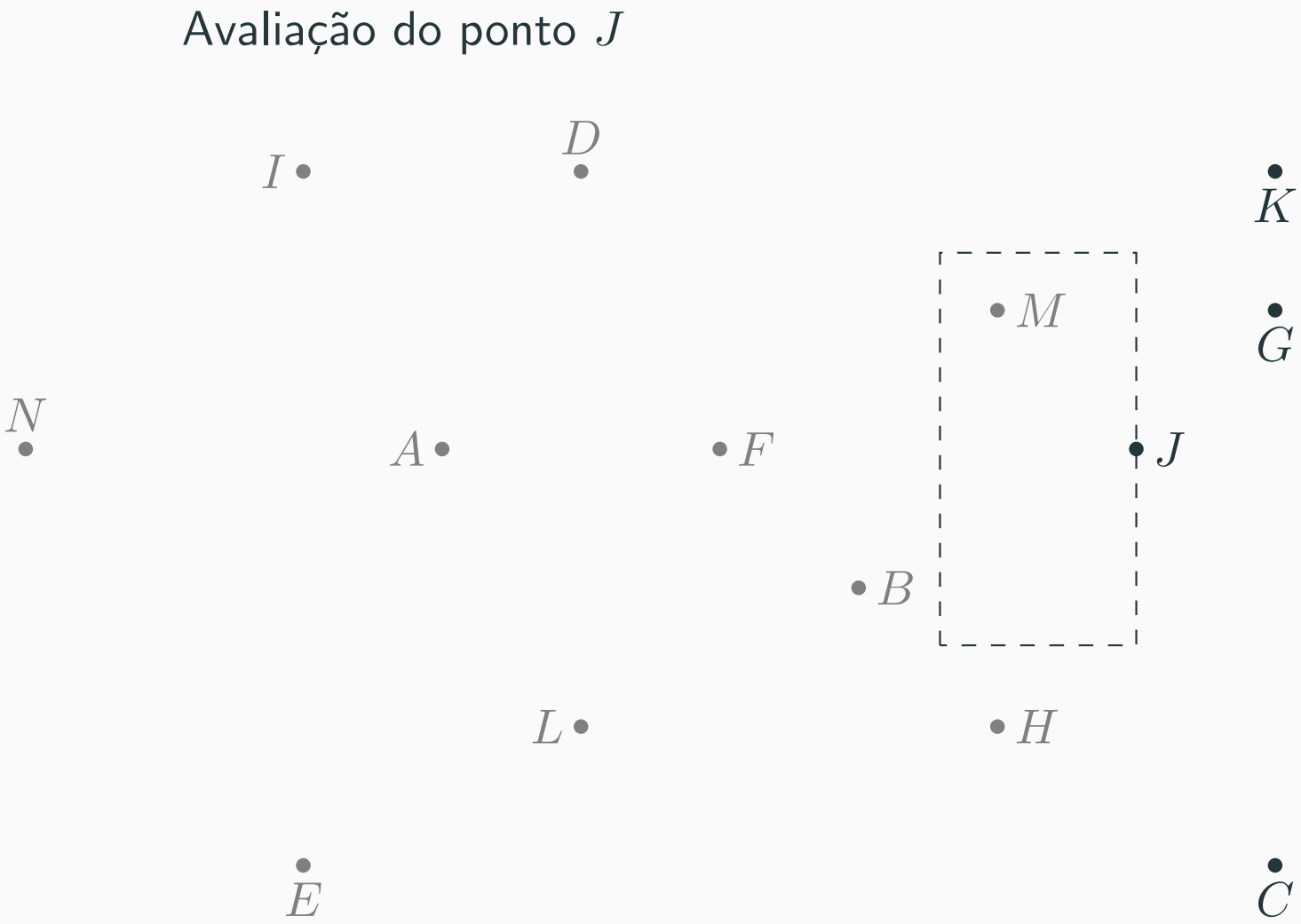
$$\text{dist}(H, B) = \text{dist}(B, F) = \mathbf{1.414213}$$



Visualização de identificação do par de pontos mais próximo

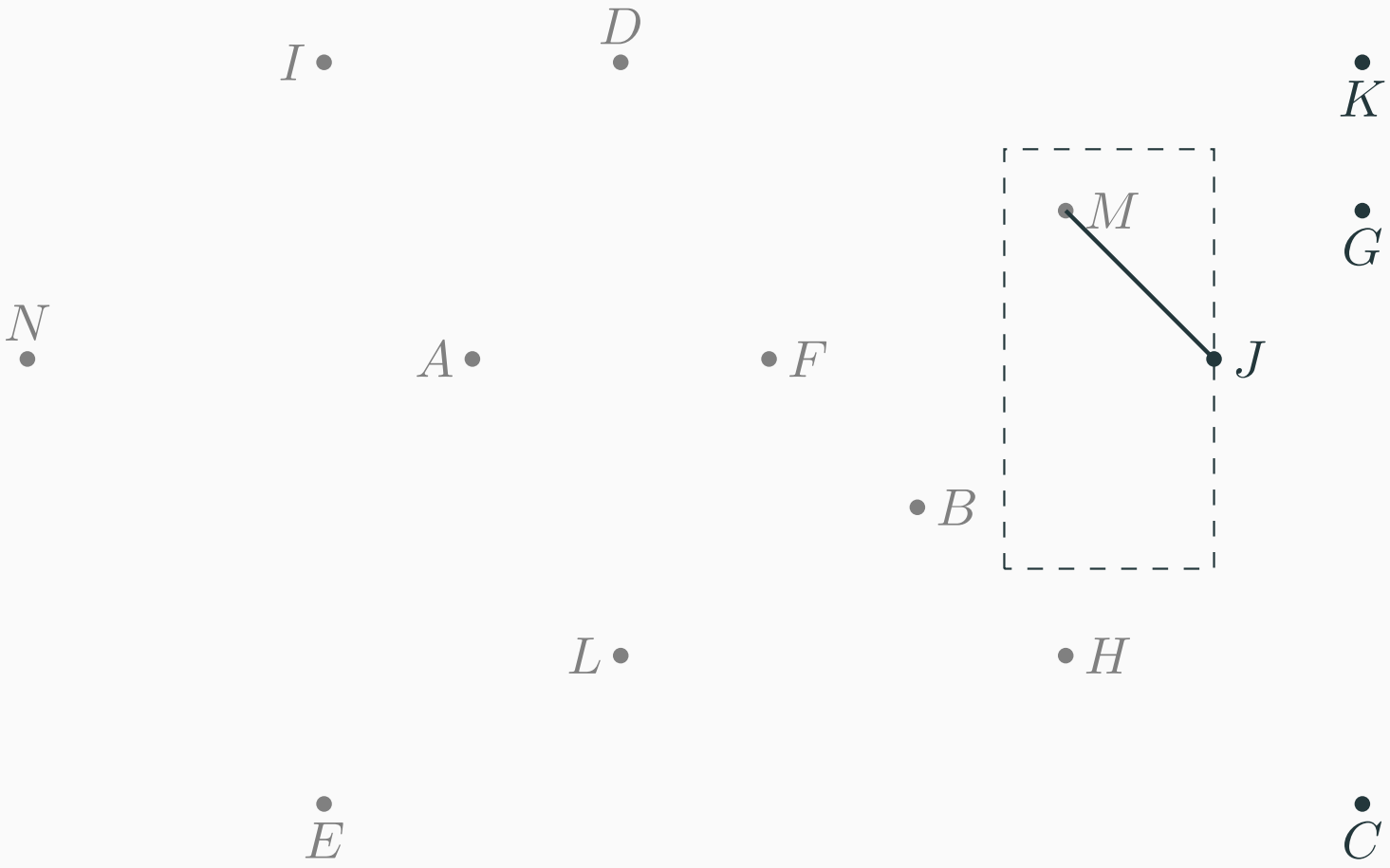


Visualização de identificação do par de pontos mais próximo



Visualização de identificação do par de pontos mais próximo

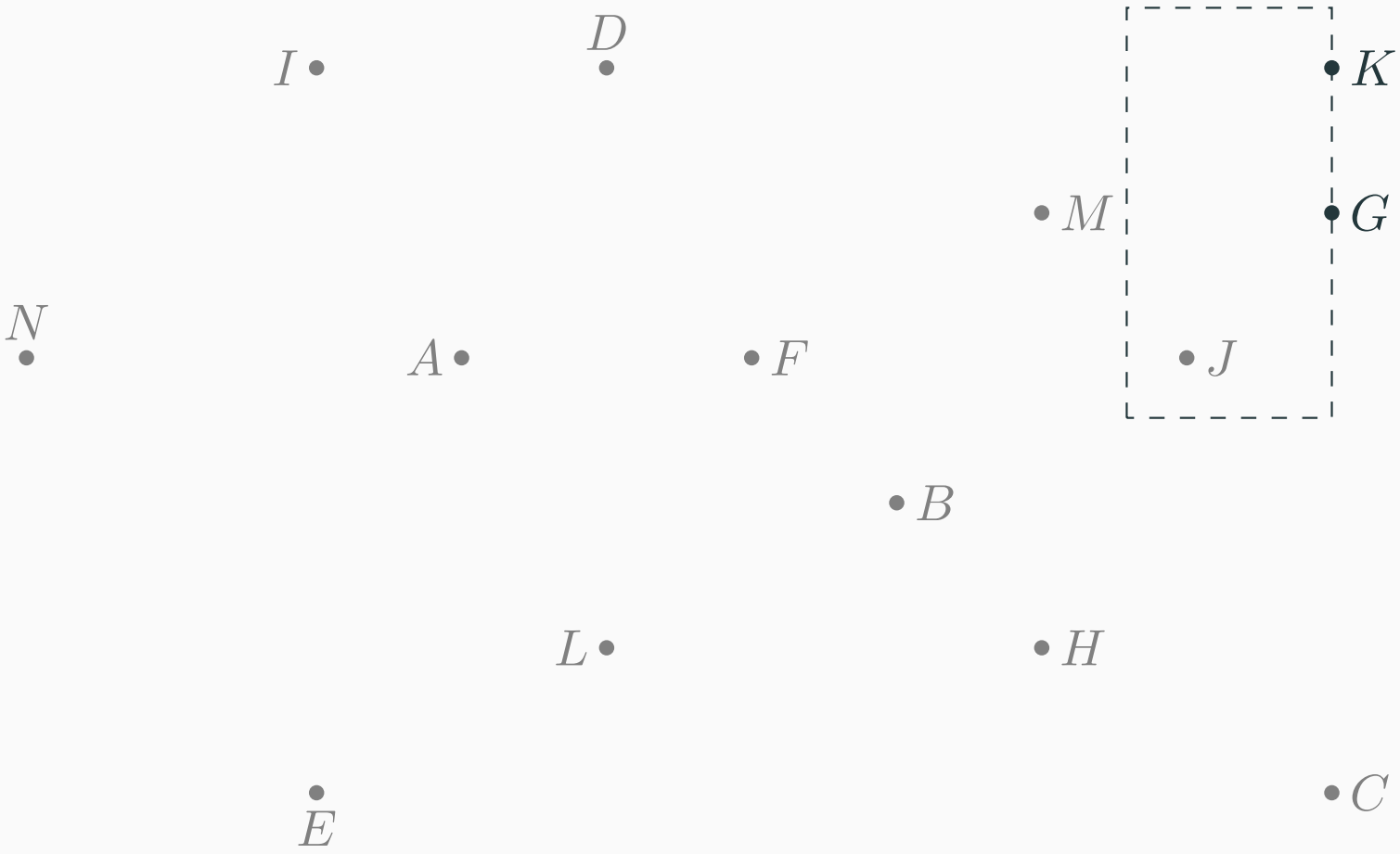
$$\mathrm{dist}(J, M) = \mathrm{dist}(B, F) = \mathbf{1.414213}$$



Visualização de identificação do par de pontos mais próximo

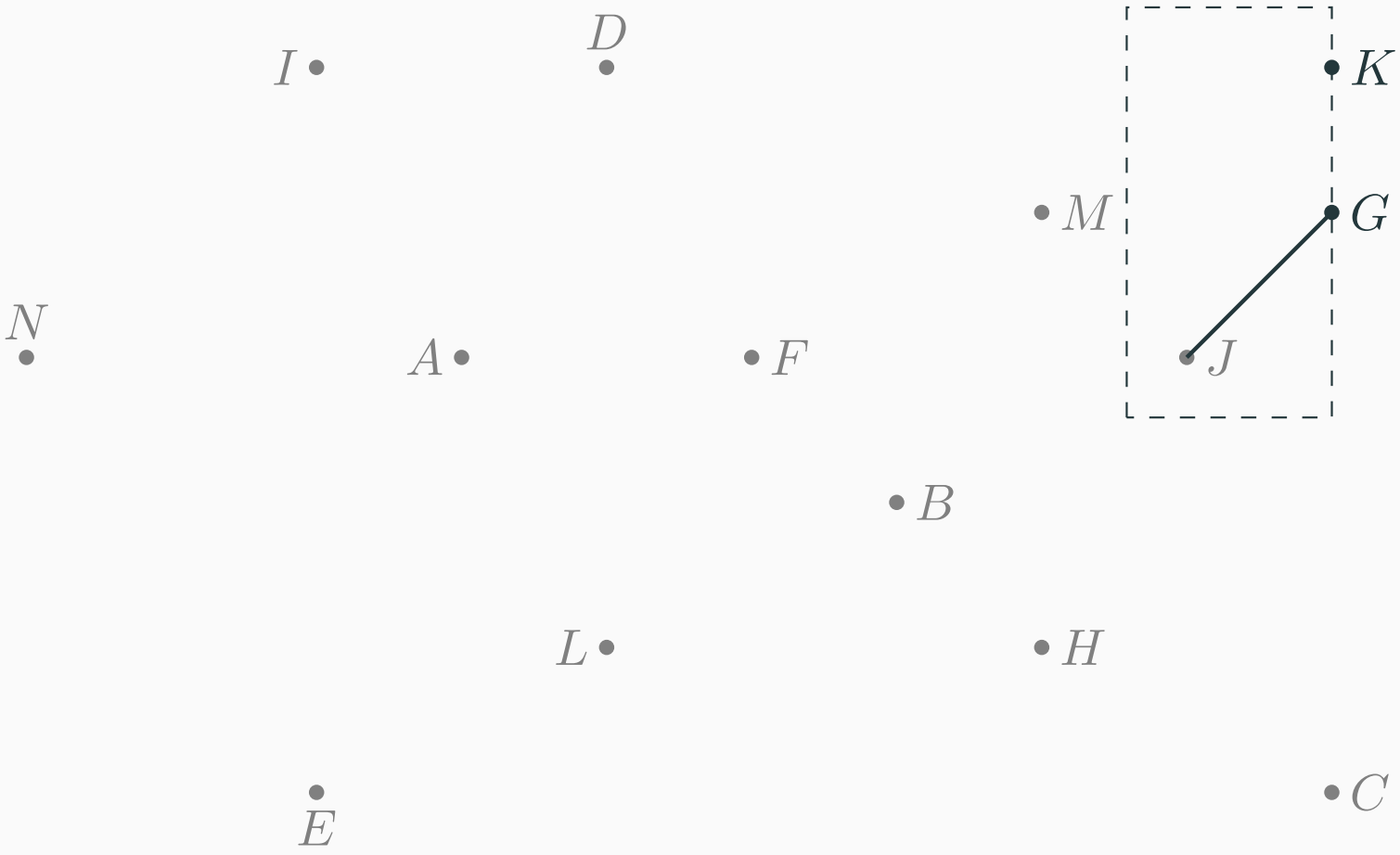
Visualização de identificação do par de pontos mais próximo

Avaliação do ponto G



Visualização de identificação do par de pontos mais próximo

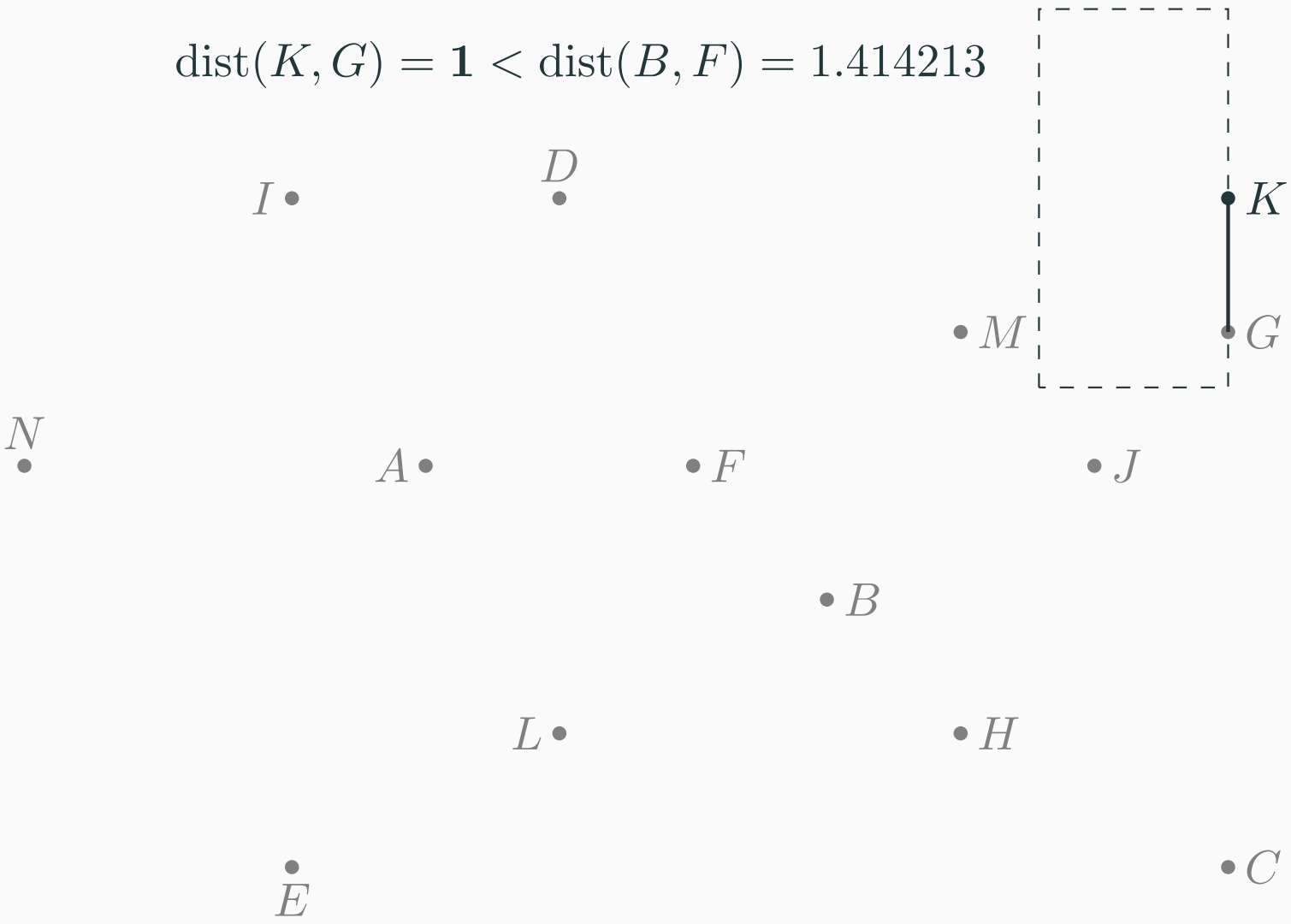
$$\mathrm{dist}(G, J) = \mathrm{dist}(B, F) = \mathbf{1.414213}$$



Visualização de identificação do par de pontos mais próximo

Visualização de identificação do par de pontos mais próximo

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



Implementação da identifacação do par mais próximo

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

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

#define x first
#define y second

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());

    // 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

```
22
23      set<ii> S;
24      S.insert(ii(ps[0].y, ps[0].x));
25      S.insert(ii(ps[1].y, ps[1].x));
26
27      for (int i = 2; i < N; ++i)
28      {
29          auto P = ps[i];
30          auto it = S.lower_bound(point(P.y - d, 0));
31
32          while (it != S.end())
33          {
34              auto Q = point(it->second, it->first);
35
36              if (Q.x < P.x - d)
37              {
38                  it = S.erase(it);
39                  continue;
40              }
41
```

Implementação da identifacação do par mais próximo

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

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 $r \cap s \neq \emptyset$
- 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

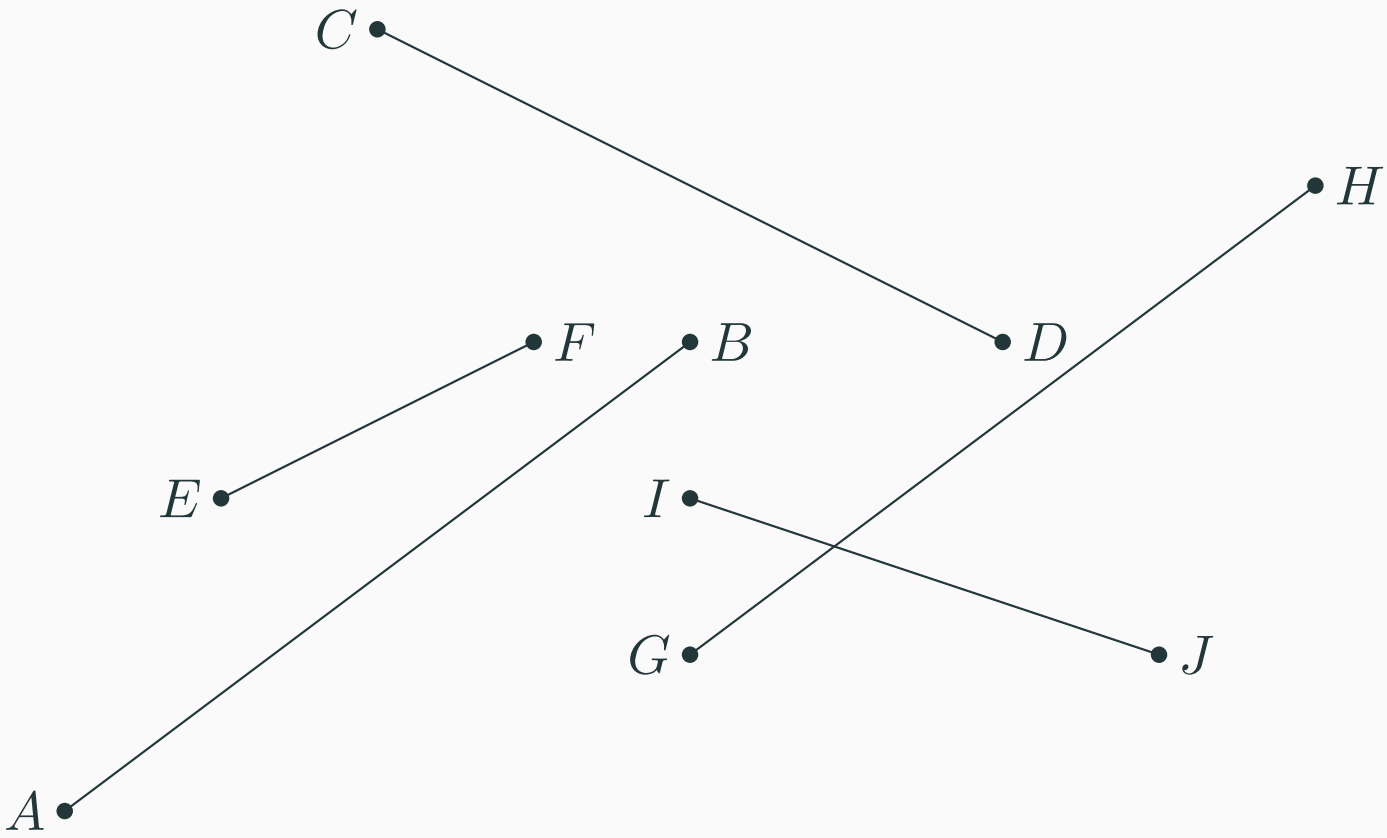
Algoritmo de Shamos-Hoey

- 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 $O(N)$
- 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 $x = x_e$

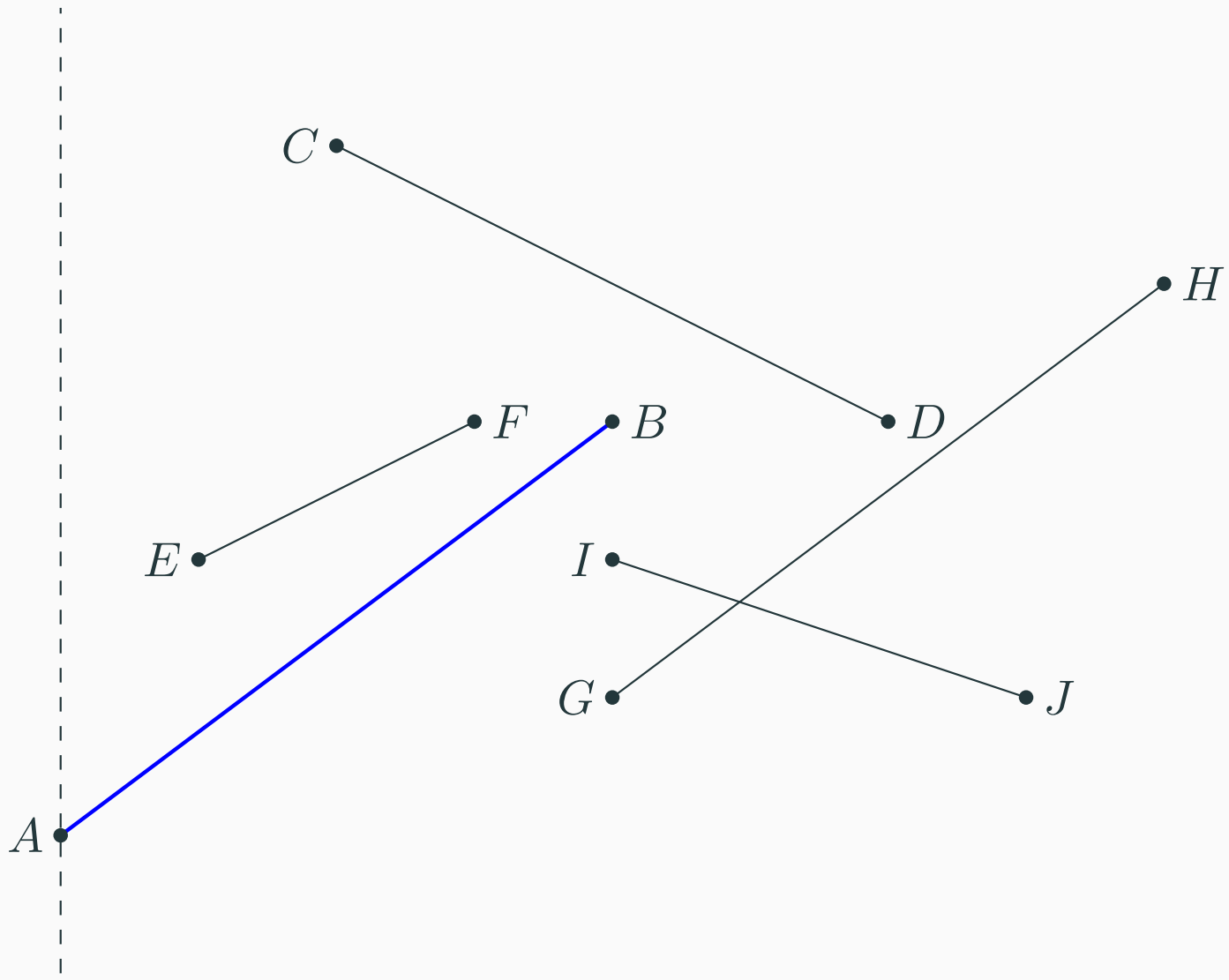
Algoritmo de Shamos-Hoey

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

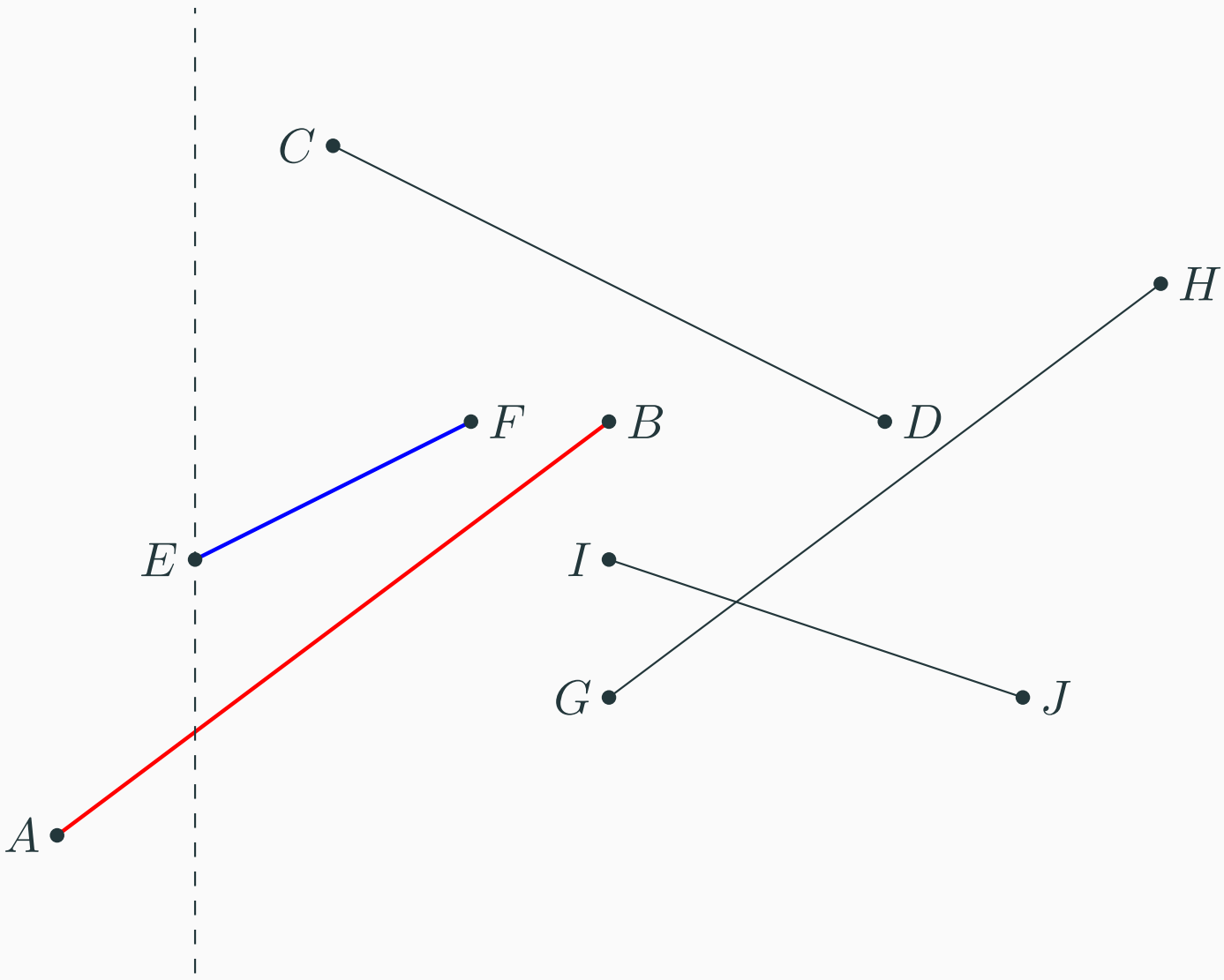
Visualização de identificação de interseção de segmentos



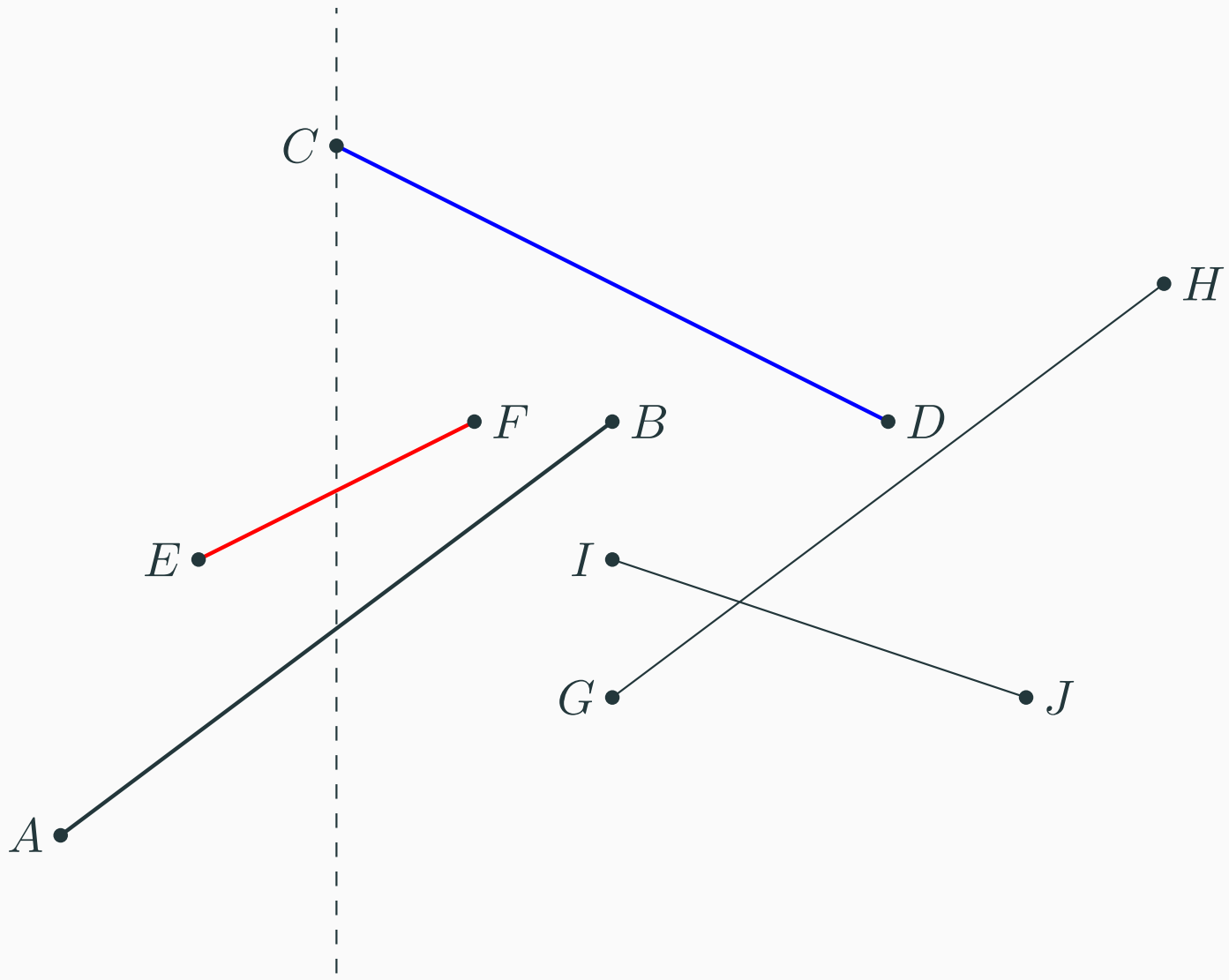
Visualização de identificação de interseção de segmentos



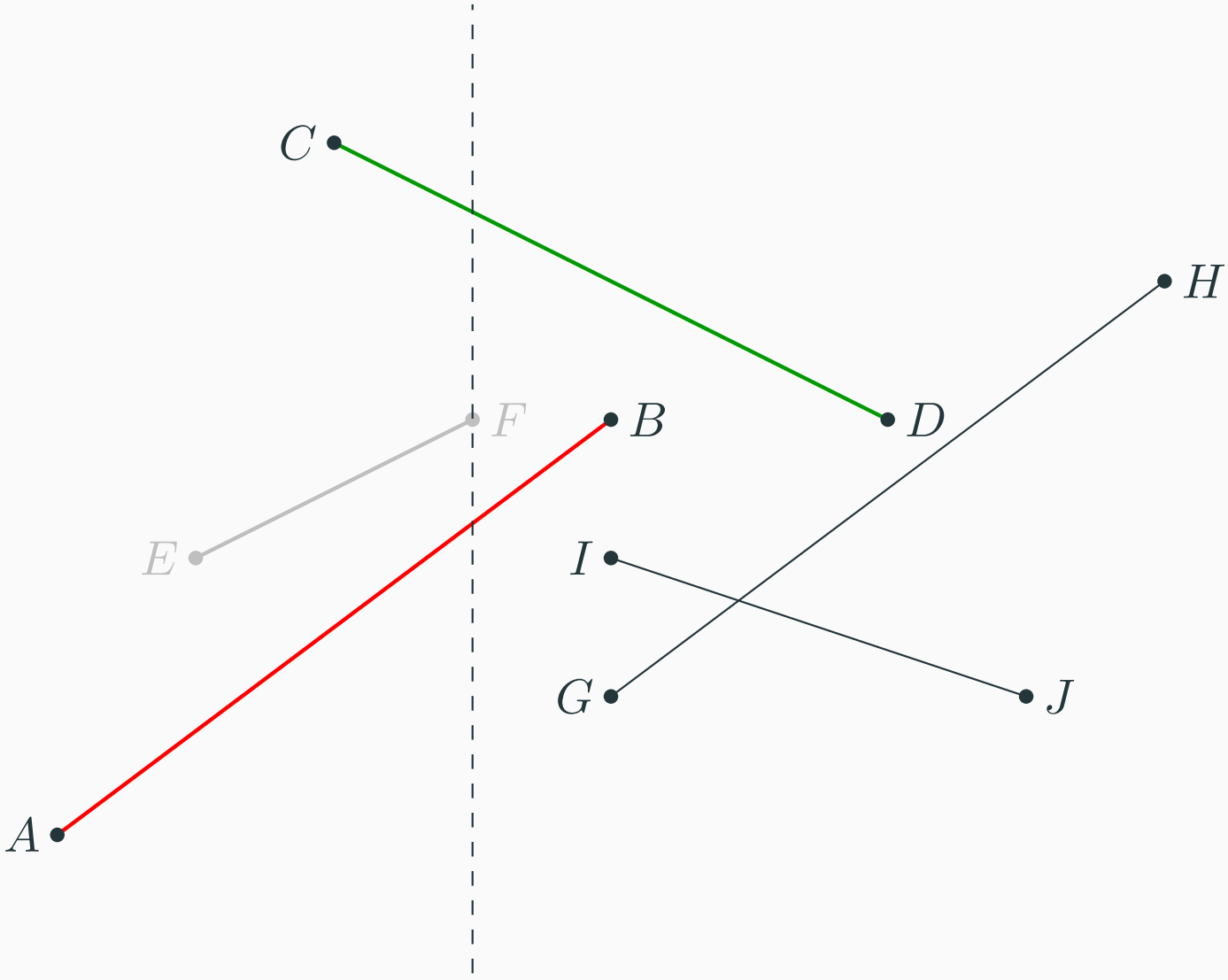
Visualização de identificação de interseção de segmentos



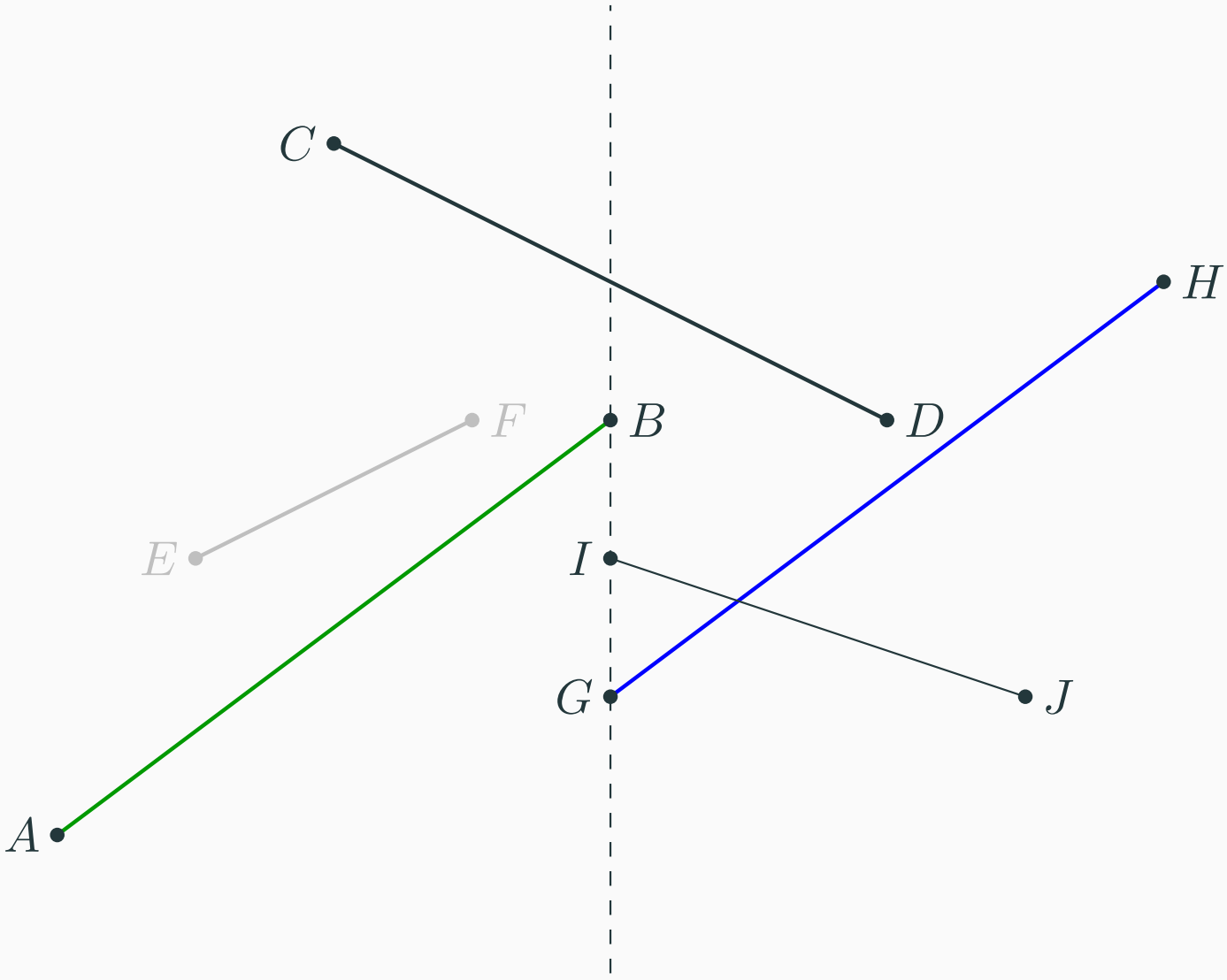
Visualização de identificação de interseção de segmentos



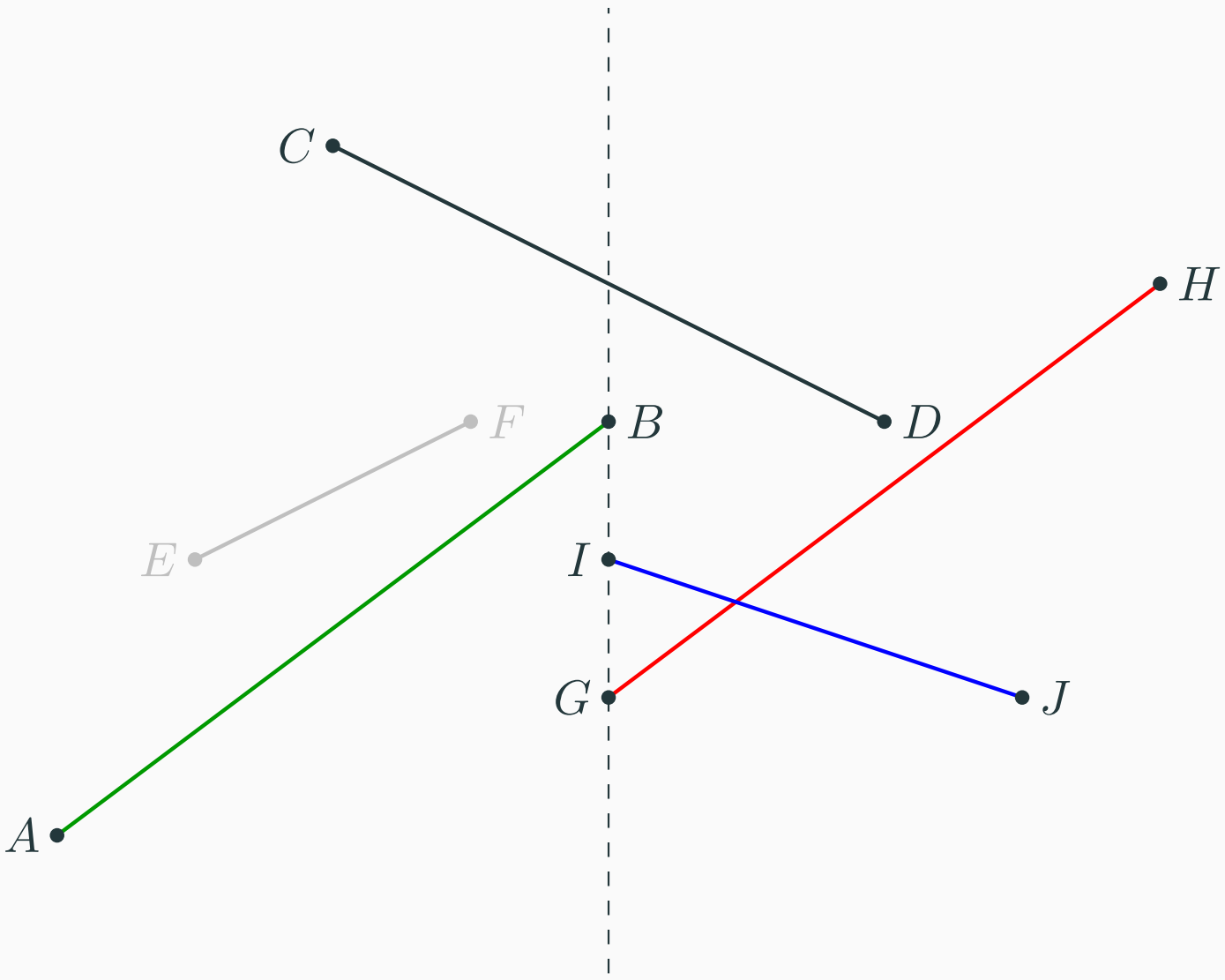
Visualização de identificação de interseção de segmentos



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
29     {
30         return x == P.x and y == P.y;
31     }
32 };
33 
34 template<typename T>
35 T D(const Point<T>& P, const Point<T>& Q, const Point<T>& R)
36 {
37     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);
38 }
```

Implementação do algoritmo de Shamos-Hoey

```
40 template<typename T>
41 struct Segment
42 {
43     T a, b, c;
44     Point<T> A, B;
45 
46     Segment(const Point<T>& P, const Point<T>& Q)
47         : a(P.y - Q.y), b (Q.x - P.x), c(P.x*Q.y - Q.x*P.y), A(P), B(Q)
48     {
49         sweep_x = -1;
50     }
51 
52     bool operator<(const Segment& line) const
53     {
54         return (-a*sweep_x - c)*line.b < (-line.a*sweep_x -line.c)*b;
55     }
56 
57     bool intersect(const Segment& s) const
58     {
59         auto d1 = D(A, B, s.A);
60         auto d2 = D(A, B, s.B);
```

Implementação do algoritmo de Shamos-Hoey

```
61
62          if ((equals(d1, 0LL) && contains(s.A)) ||
63              (equals(d2, 0LL) && contains(s.B)))
64              return true;
65
66          auto d3 = D(s.A, s.B, A);
67          auto d4 = D(s.A, s.B, B);
68
69          if ((equals(d3, 0LL) && s.contains(A)) ||
70              (equals(d4, 0LL) && s.contains(B)))
71              return true;
72
73          return (d1 * d2 < 0) && (d3 * d4 < 0);
74      }
75
76      bool contains(const Point<T>& P) const
77      {
78          if (P == A || P == B)
79              return true;
80
```

Implementação do algoritmo de Shamos-Hoey

```
81          auto xmin = min(A.x, B.x);
82          auto xmax = max(A.x, B.x);
83          auto ymin = min(A.y, B.y);
84          auto ymax = max(A.y, B.y);
85
86          if (P.x < xmin || P.x > xmax || P.y < ymin || P.y > ymax)
87              return false;
88
89          return equals((P.y - A.y)*(B.x - A.x), (P.x - A.x)*(B.y - A.y));
90      }
91
92      static T sweep_x;
93  };
94
95  template<typename T>
96  T Segment<T>::sweep_x;
97
```

Implementação do algoritmo de Shamos-Hoey

```
98  template<typename T>
99  bool shamos_hoey(const vector<Segment<T>>& segments)
100 {
101     struct Event
102     {
103         Point<T> P;
104         size_t i;
105
106         bool operator<(const Event& e) const { return P < e.P; }
107     };
108
109     vector<Event> events;
110
111     for (size_t i = 0; i < segments.size(); ++i)
112     {
113         events.push_back({ segments[i].A, i });
114         events.push_back({ segments[i].B, i });
115     }
116
117     sort(events.begin(), events.end());
118     set<Segment<T>> sl;
```

Implementação do algoritmo de Shamos-Hoey

```
120     for (const auto& e : events)
121     {
122         auto s = segments[e.i];
123         Segment<T>::sweep_x = e.P.x;
124 
125         if (e.P == s.A)
126         {
127             sl.insert(s);
128 
129             auto it = sl.find(s);
130 
131             if (it != sl.begin())
132             {
133                 auto L = *prev(it);
134 
135                 if (s.intersect(L)) return true;
136             }
137 
138             if (next(it) != sl.end())
139             {
140                 auto U = *next(it);
```

Implementação do algoritmo de Shamos-Hoey

```
141
142                 if (s.intersect(U)) return true;
143             }
144         } else
145         {
146             auto it = sl.find(s);
147
148             if (it != sl.begin() and it != sl.end())
149             {
150                 auto L = *prev(it);
151                 auto U = *next(it);
152
153                 if (L.intersect(U)) return true;
154             }
155
156             sl.erase(it);
157         }
158     }
159
160     return false;
161 }
```

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;
}

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

Implementação do algoritmo de busca completa

```
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
29      {
30          return x == P.x and y == P.y;
31      }
32  };
33
34  template<typename T>
35  struct Segment
36  {
37      T a, b, c;
38      Point<T> A, B;
39
40      Segment(const Point<T>& P, const Point<T>& Q)
41          : a(P.y - Q.y), b (Q.x - P.x), c(P.x*Q.y - Q.x*P.y), A(P), B(Q)
42      {
43      }
```

Implementação do algoritmo de busca completa

```
44
45      optional<Point<T>> intersection(const Segment& s)
46      {
47          auto det = a * s.b - b * s.a;
48
49          if (not equals(det, 0.0))     // Concorrentes
50          {
51              auto x = (-c * s.b + s.c * b) / det;
52              auto y = (-s.c * a + c * s.a) / det;
53
54              if (min(A.x, B.x) <= x and x <= max(A.x, B.x) and
55                  min(s.A.x, s.B.x) <= x and x <= max(s.A.x, s.B.x))
56              {
57                  return Point<T> { x, y };
58              }
59          }
60
61          return { };
62      }
63  };
64
```

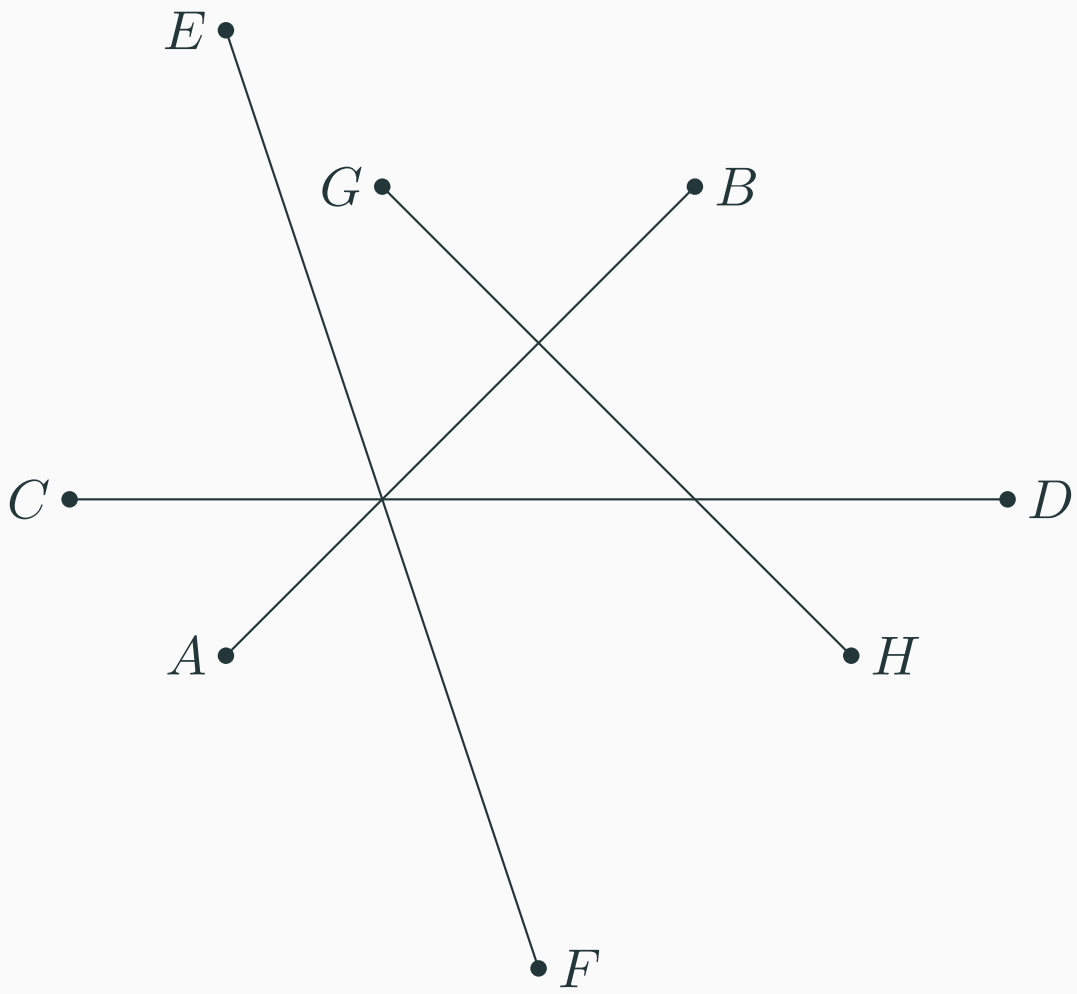
Implementação do algoritmo de busca completa

```
65 template<typename T>
66 set<Point<T>> intersections(int N, const vector<Segment<T>>& segments)
67 {
68     set<Point<T>> ans;
69 
70     for (int i = 0; i < N; ++i)
71     {
72         auto s = segments[i];
73 
74         for (int j = i + 1; j < N; ++j)
75         {
76             auto r = segments[j];
77             auto P = s.intersection(r);
78 
79             if (P)
80                 ans.insert(P.value());
81         }
82     }
83 
84     return ans;
85 }
```

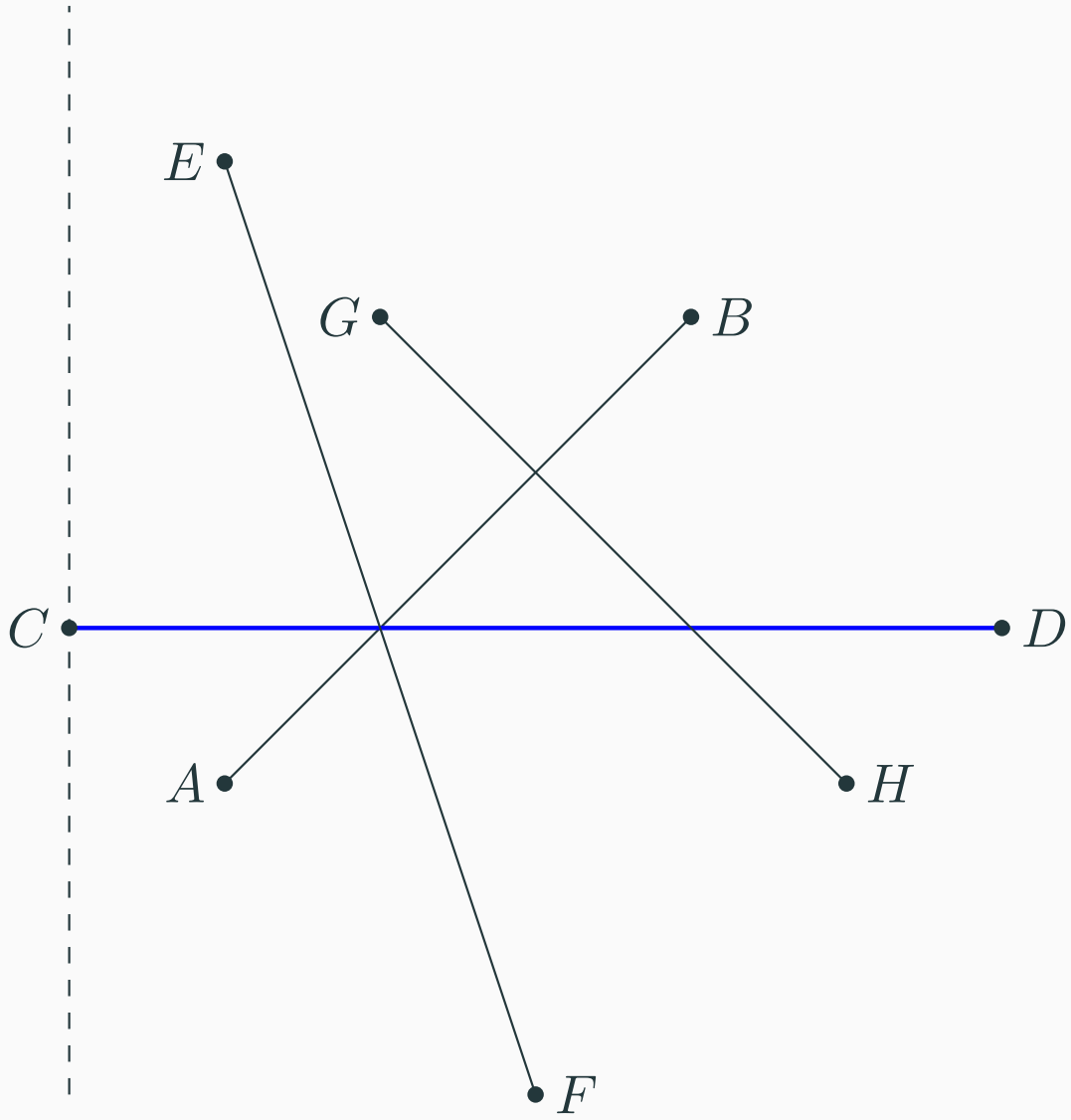
Algoritmo de Bentley-Ottman

- 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

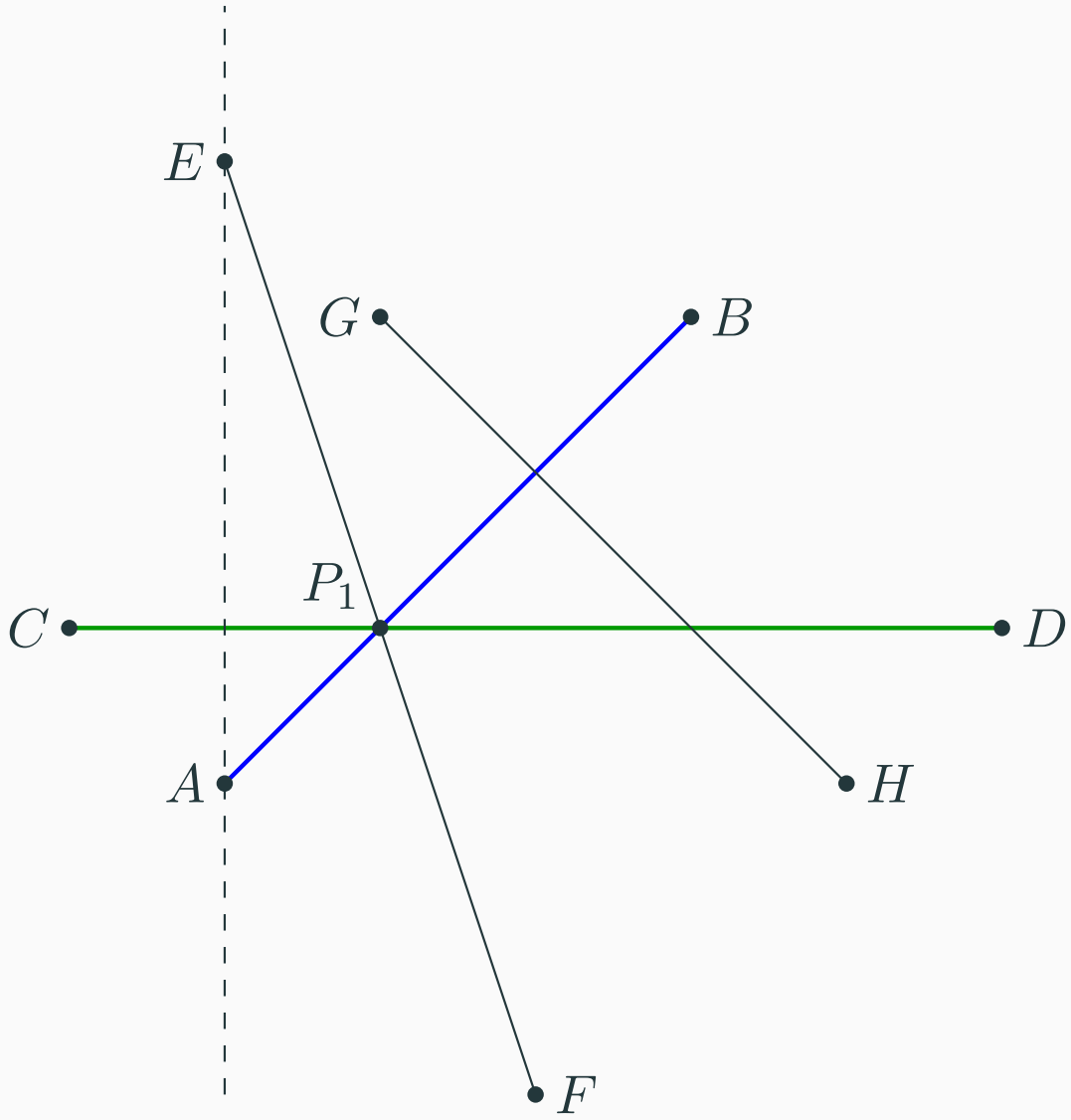
Visualização da contagem das interseções entres segmentos



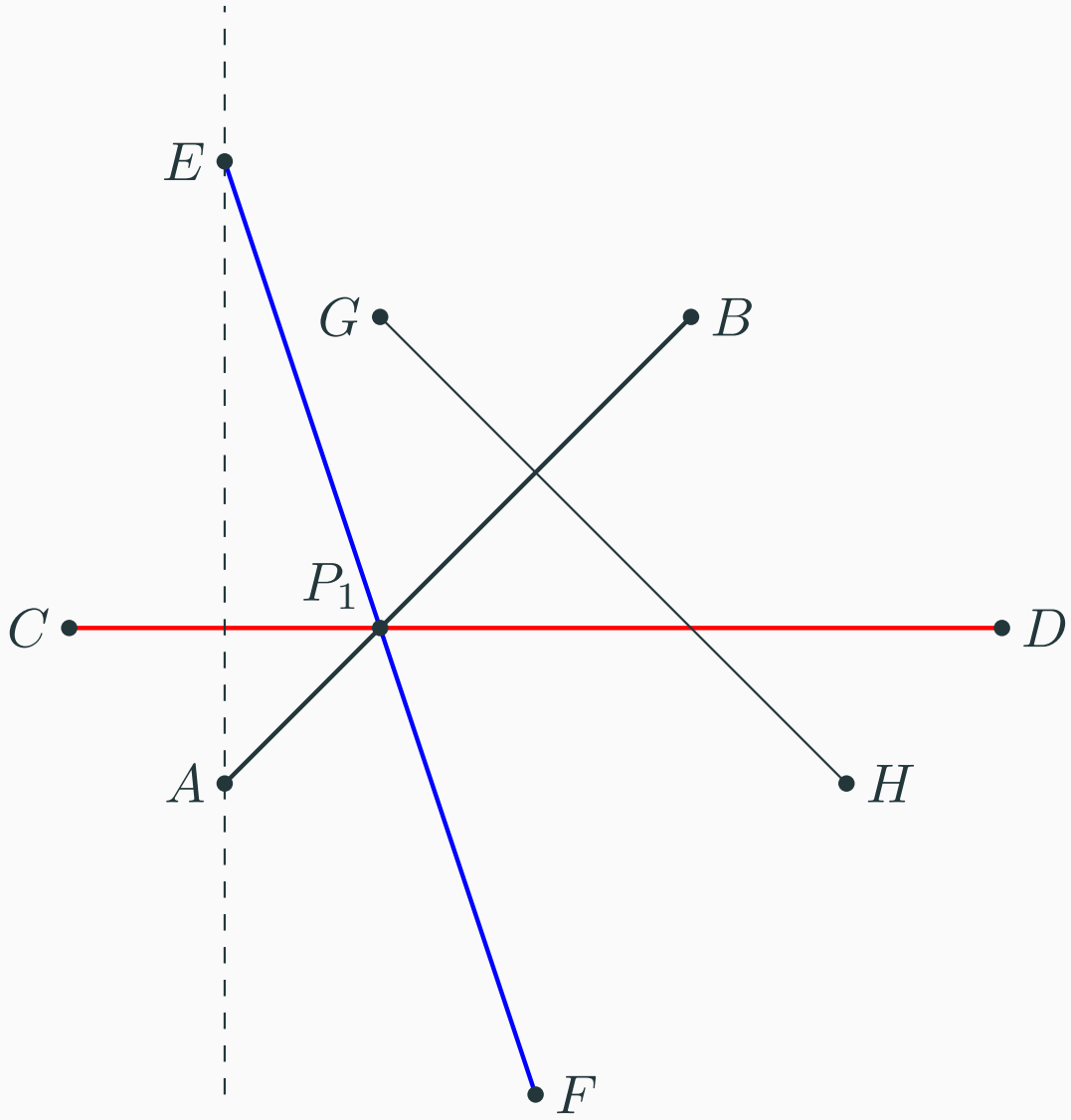
Visualização da contagem das interseções entres segmentos



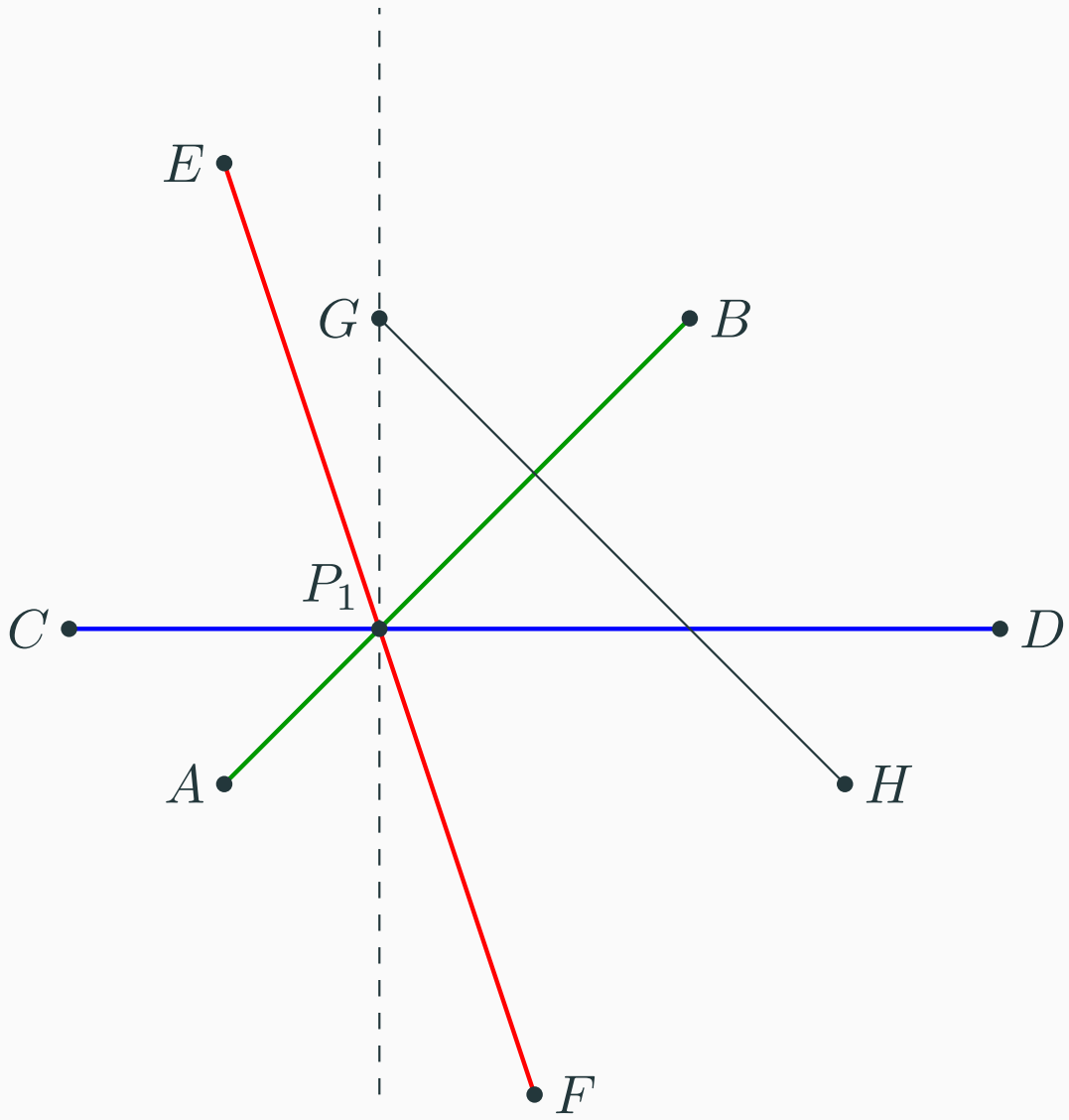
Visualização da contagem das interseções entres segmentos



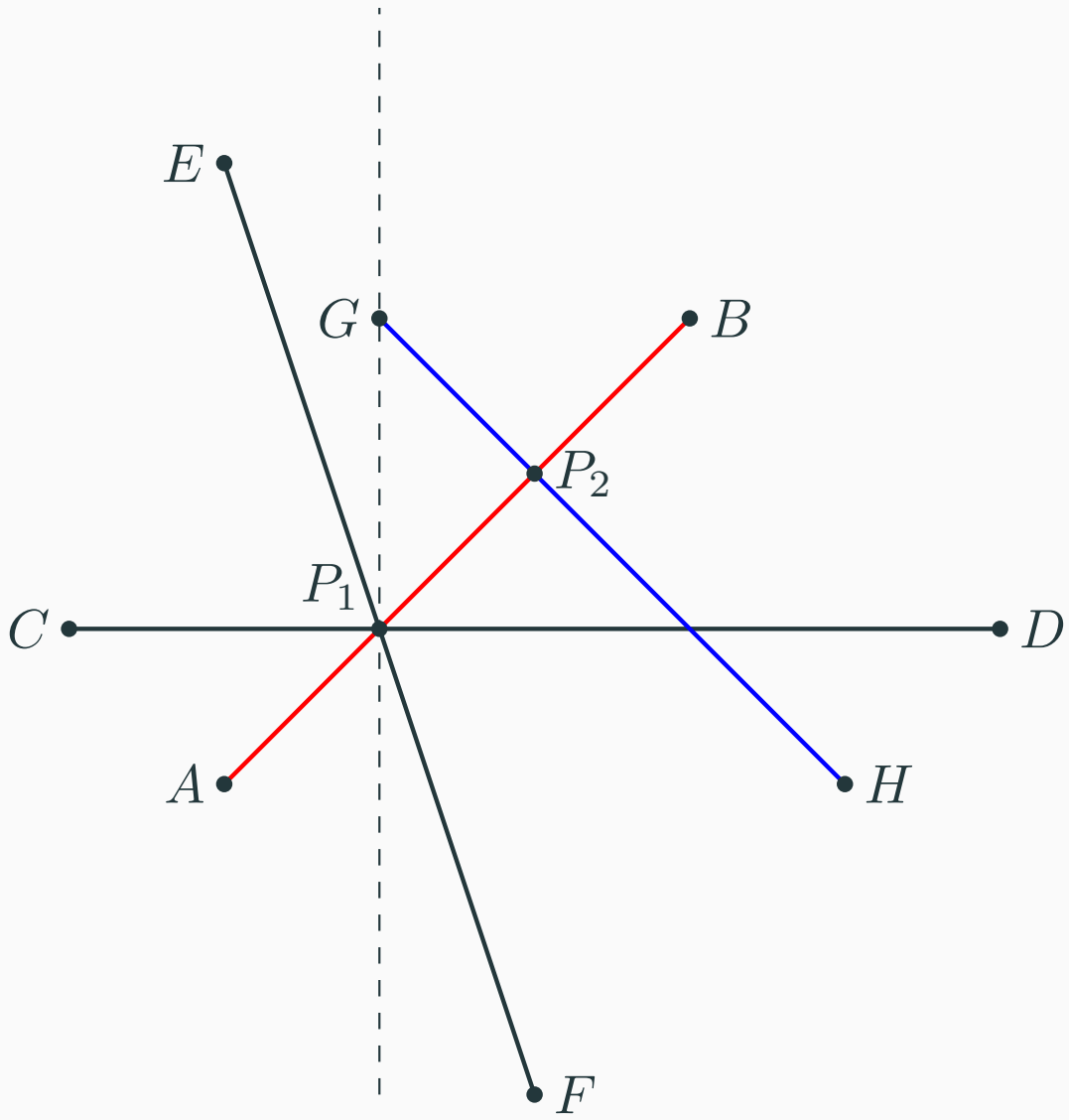
Visualização da contagem das interseções entres segmentos



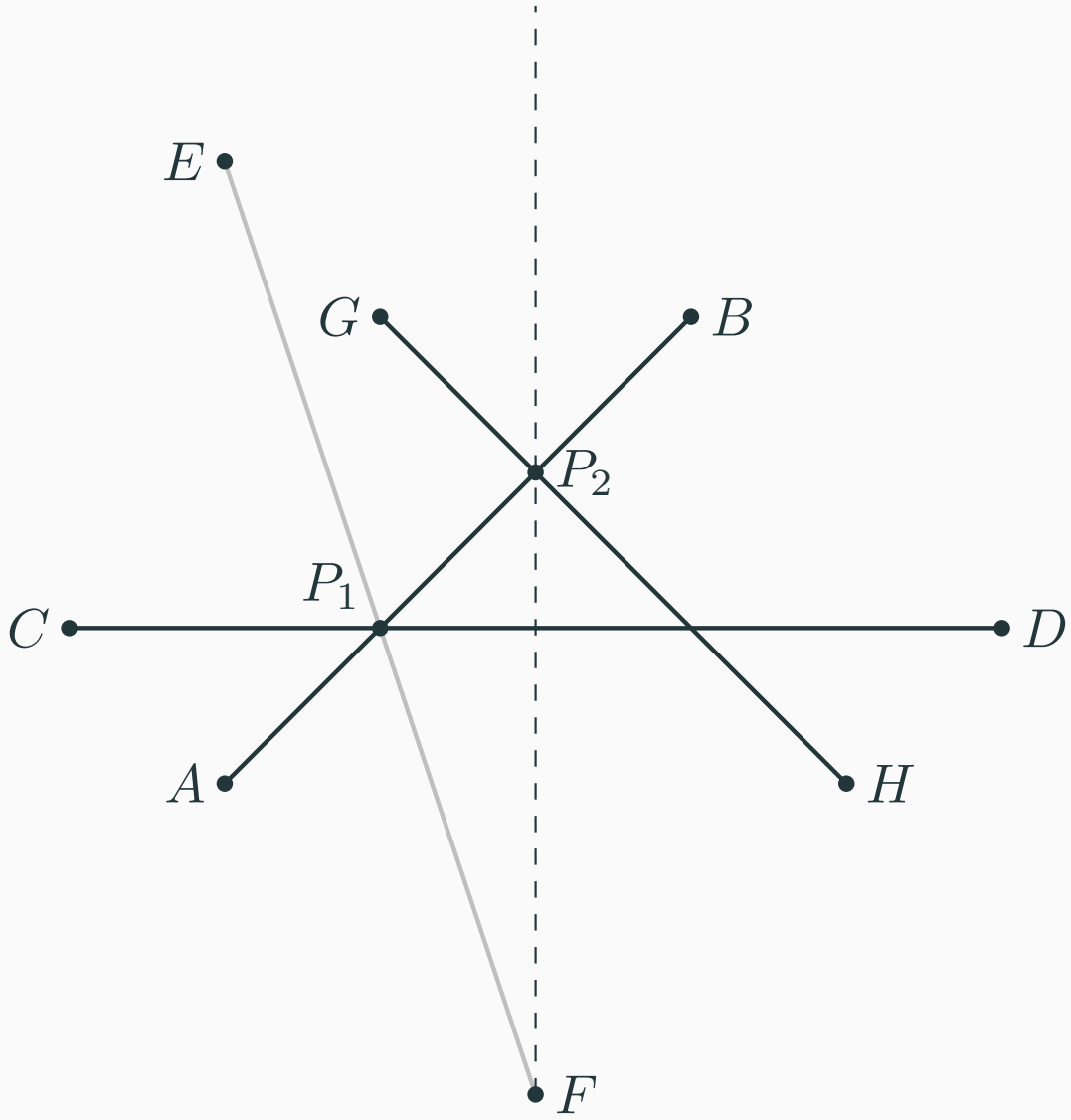
Visualização da contagem das interseções entres segmentos



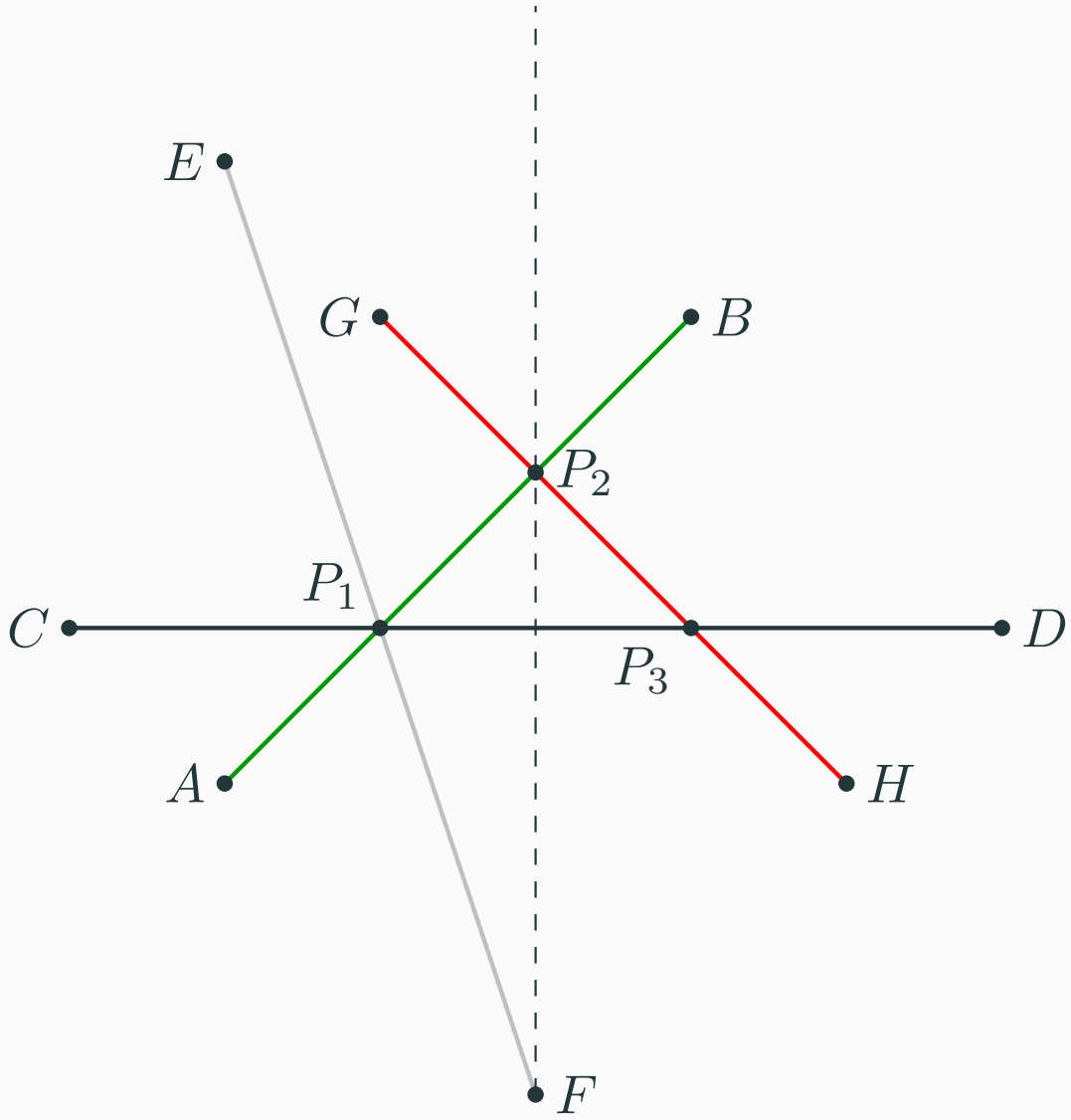
Visualização da contagem das interseções entres segmentos



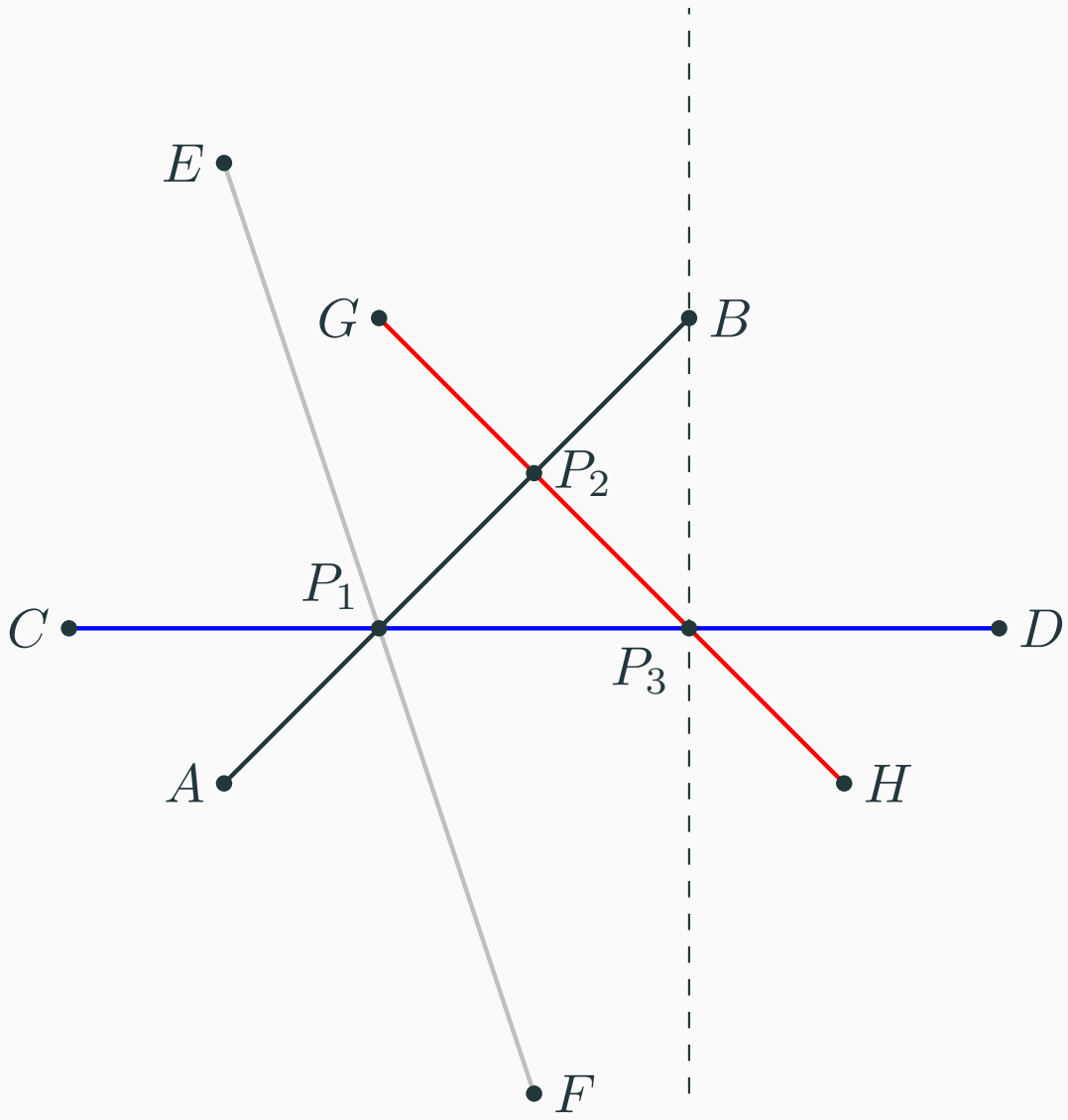
Visualização da contagem das interseções entres segmentos



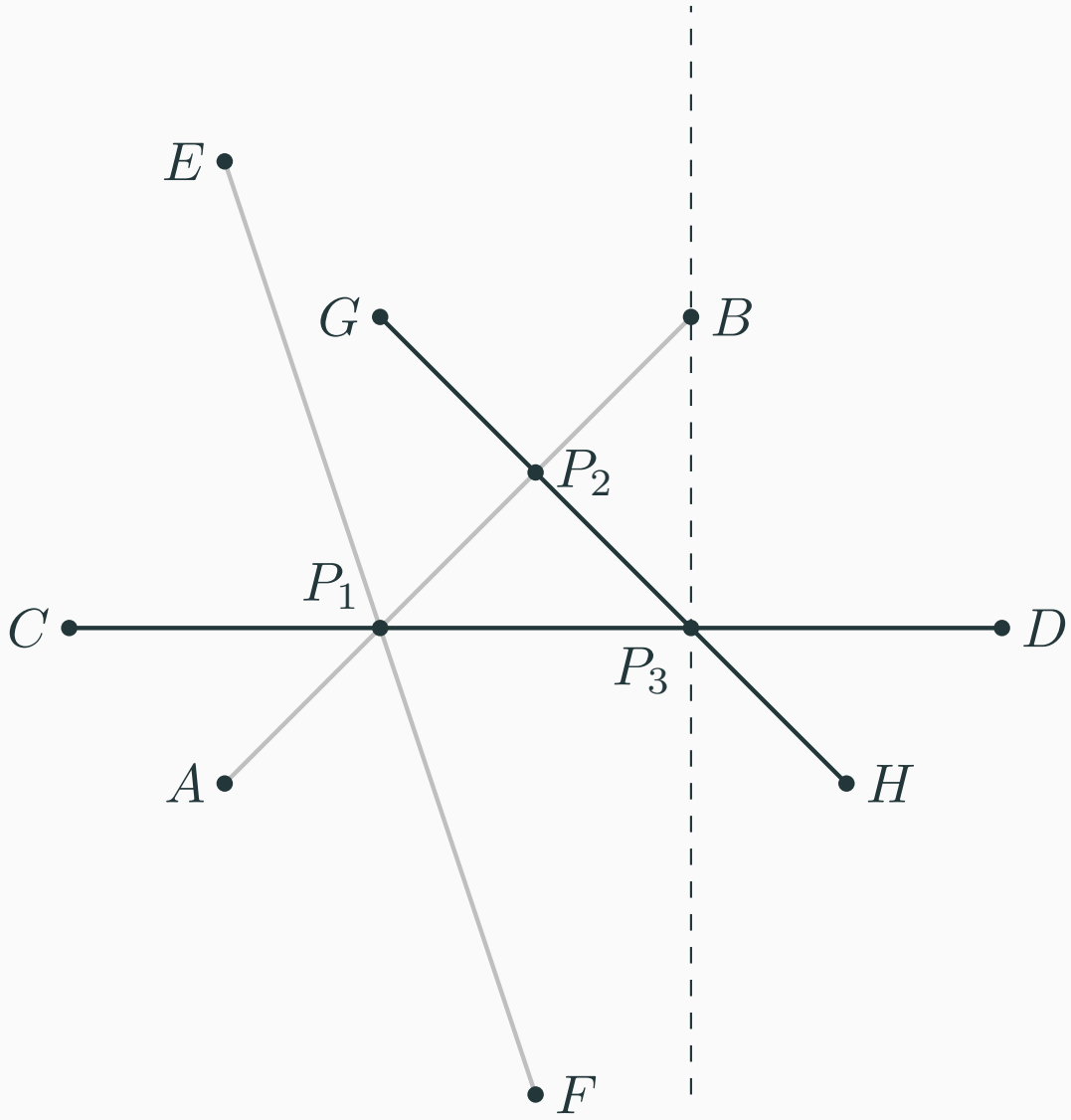
Visualização da contagem das interseções entres segmentos



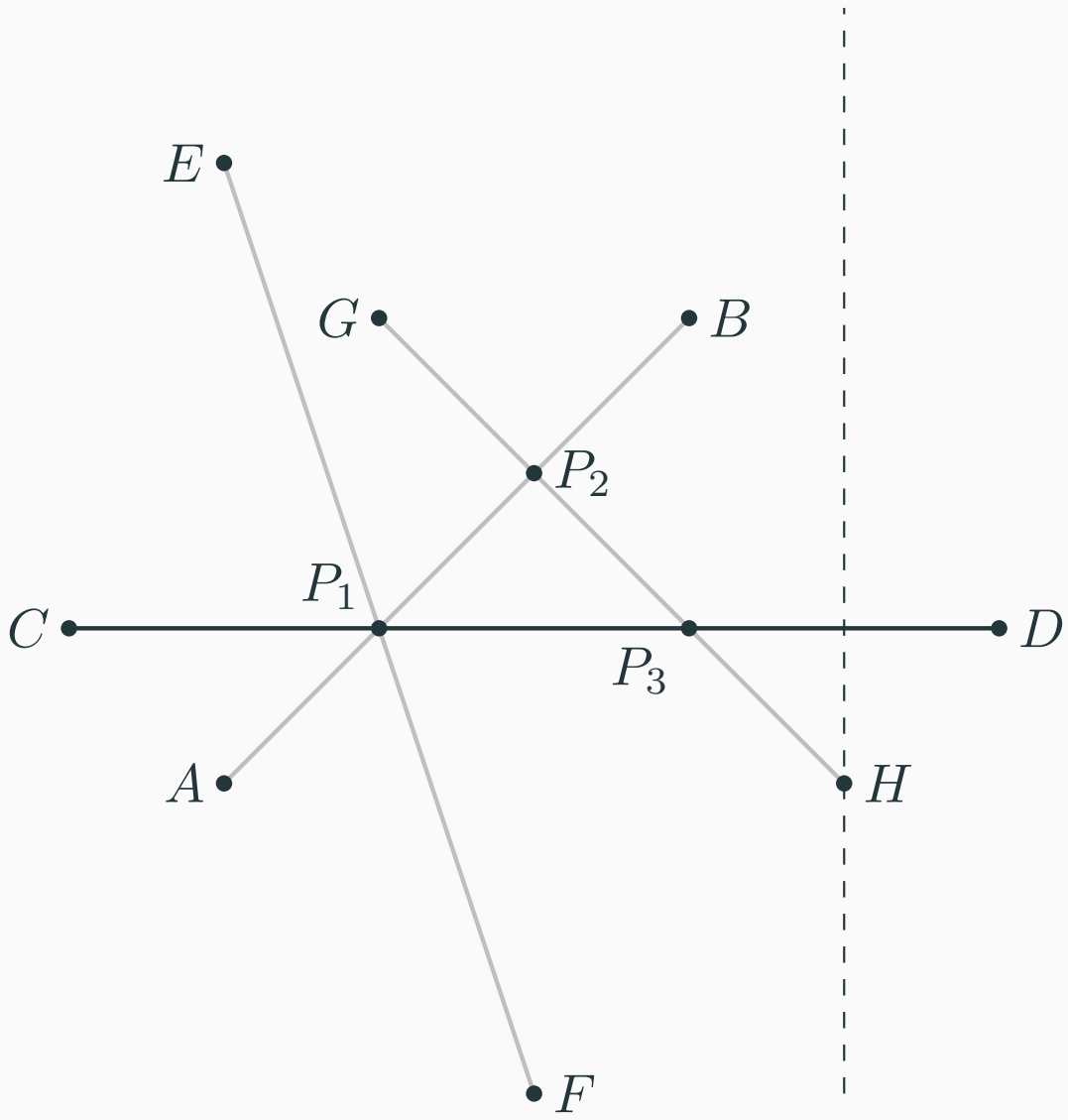
Visualização da contagem das interseções entres segmentos



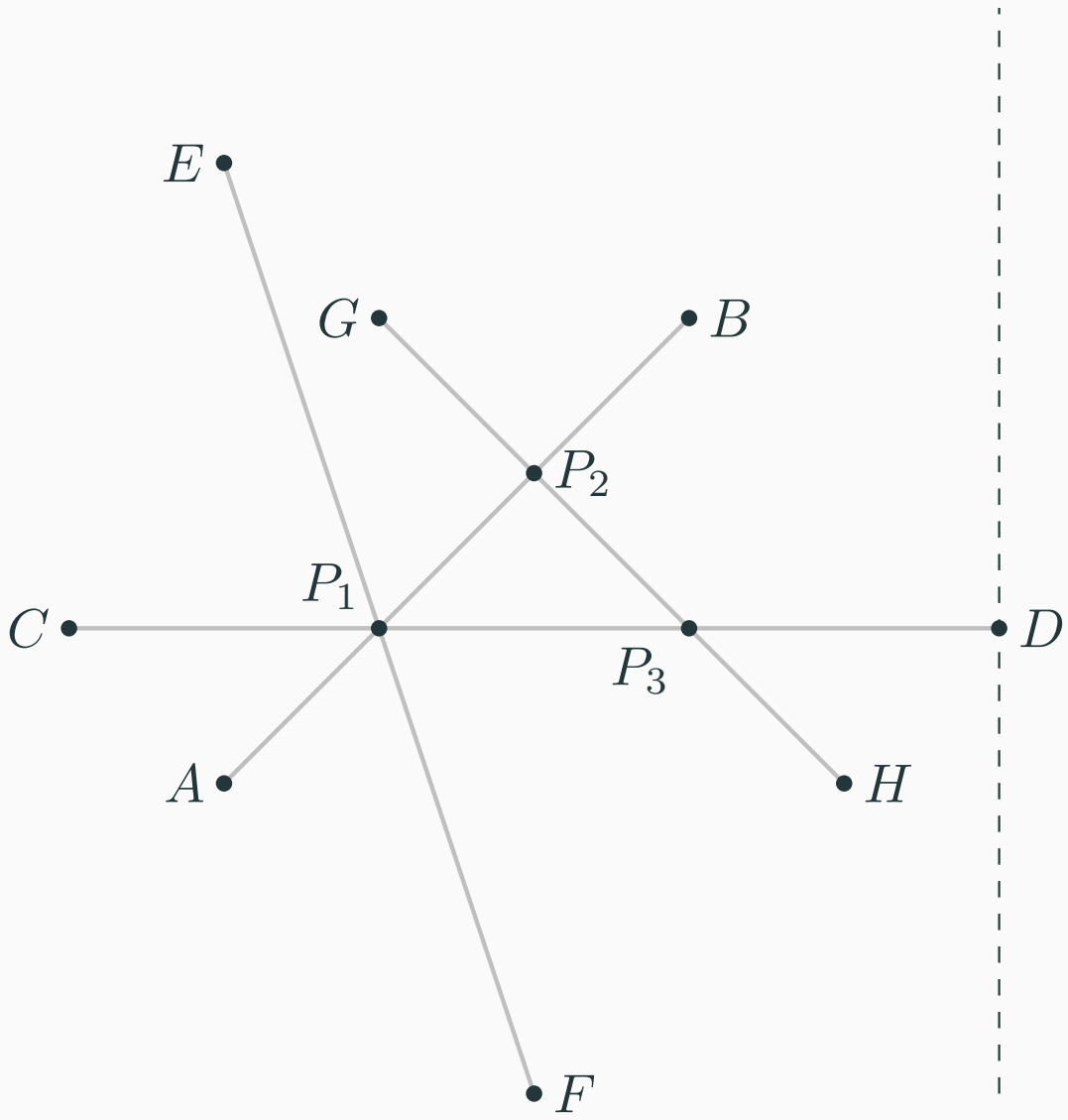
Visualização da contagem das interseções entres segmentos



Visualização da contagem das interseções entres segmentos



Visualização da contagem das interseções entres segmentos



Implementação do algoritmo de Bentley-Ottman

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  bool equals(double a, double b)
6  {
7      const double EPS { 1e-6 };
8
9      return fabs(a - b) < EPS;
10 }
11
12 struct Point
13 {
14     double x, y;
15
16     bool operator<(const Point& P) const
17     {
18         return x != P.x ? x < P.x : y < P.y;
19     }
20
```

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),
40            A(P), B(Q), idx(i) { }
41
```

Implementação do algoritmo de Bentley-Ottman

```
42      bool operator<(const Segment& s) const
43      {
44          return (-a*sweep_x - c)*s.b < (-s.a*sweep_x -s.c)*b;
45      }
46
47      optional<Point> intersection(const Segment& s) const
48      {
49          auto det = a * s.b - b * s.a;
50
51          if (not equals(det, 0.0))     // Concorrentes
52          {
53              auto x = (-c * s.b + s.c * b) / det;
54              auto y = (-s.c * a + c * s.a) / det;
55
56              if (min(A.x, B.x) <= x and x <= max(A.x, B.x) and
57                  min(s.A.x, s.B.x) <= x and x <= max(s.A.x, s.B.x))
58              {
59                  return Point { x, y };
60              }
61          }
62
```

Implementação do algoritmo de Bentley-Ottman

```
63          return { };
64      }
65
66      static double sweep_x;
67  };
68
69  double Segment::sweep_x;
70
71  struct Event
72  {
73      enum Type { OPEN, INTERSECTION, CLOSE };
74
75      Point P;
76      Type type;
77      size_t i;
78
79      bool operator<(const Event& e) const
80      {
81          if (P != e.P)
82              return e.P < P;
83
```

Implementação do algoritmo de Bentley-Ottman

```
84          if (type != e.type)
85              return type > e.type;
86
87          return i > e.i;
88      }
89  };
90
91  void add_neighbor_intersections(const Segment& s, const set<Segment>& sl,
92      set<Point>& ans, priority_queue<Event>& events)
93  {
94      // TODO: garantir que a busca identifique unicamente o elemento s,
95      // através do ajuste fino da variável Segment::sweep_x
96      auto it = sl.find(s);
97
98      if (it != sl.begin())
99      {
100         auto L = *prev(it);
101         auto P = s.intersection(L);
102
```

Implementação do algoritmo de Bentley-Ottman

```
103         if (P and ans.count(P.value()) == 0)
104         {
105             events.push(Event { P.value(), Event::INTERSECTION, s.idx } );
106             ans.insert(P.value());
107         }
108     }
109
110     if (next(it) != sl.end())
111     {
112         auto U = *next(it);
113         auto P = s.intersection(U);
114
115         if (P and ans.count(P.value()) == 0)
116         {
117             events.push(Event { P.value(), Event::INTERSECTION, s.idx } );
118             ans.insert(P.value());
119         }
120     }
121 }
122
```

Implementação do algoritmo de Bentley-Ottman

```
123 set<Point> bentley_ottman(vector<Segment>& segments)
124 {
125     set<Point> ans;
126     priority_queue<Event> events;
127 
128     for (size_t i = 0; i < segments.size(); ++i)
129     {
130         events.push(Event { segments[i].A, Event::OPEN, i });
131         events.push(Event { segments[i].B, Event::CLOSE, i });
132     }
133 
134     set<Segment> sl;
135 
136     while (not events.empty())
137     {
138         auto e = events.top();
139         events.pop();
140 
141         Segment::sweep_x = e.P.x;
142 
```

Implementação do algoritmo de Bentley-Ottman

```
143         switch (e.type) {
144         case Event::OPEN:
145         {
146             auto s = segments[e.i];
147             sl.insert(s);
148 
149             add_neighbor_intersections(s, sl, ans, events);
150         }
151         break;
152 
153         case Event::CLOSE:
154         {
155             auto s = segments[e.i];
156             auto it = sl.find(s);       // TODO: aqui também
157 
158             if (it != sl.begin() and it != sl.end())
159             {
160                 auto L = *prev(it);
161                 auto U = *next(it);
162                 auto P = L.intersection(U);
163 
```

Implementação do algoritmo de Bentley-Ottman

```
164                 if (P and ans.count(P.value()) == 0)
165                     events.push( Event { P.value(), Event::INTERSECTION, L.idx } )
166             }
167
168             sl.erase(it);
169         }
170         break;
171
172         default:
173             auto r = segments[e.i];
174             auto p = sl.equal_range(r);
175
176             vector<Segment> range(p.first, p.second);
177
178             // Remove os segmentos que se interceptam
179             sl.erase(p.first, p.second);
180
181             // Reinsere os segmentos
182             Segment::sweep_x += 0.1;
183
184             sl.insert(range.begin(), range.end());
```

Implementação do algoritmo de Bentley-Ottman

```
185
186             // Procura interseções com os novos vizinhos
187             for (const auto& s : range)
188                 add_neighbor_intersections(s, sl, ans, events);
189         }
190     }
191
192     return ans;
193 }
194
```

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.