Geometria Computacional

Tópicos Avançados em Algoritmos

Prof. Hamilton José Brumatto

Geometria Computacional

Este texto tem o objetivo de descrever os códidos de geometria computacional oferecidos durante a disciplina. A geometria computacional se encarrega em parametrizar pontos, linhas, curvas, arcos, vetores, polígonos, ... e tratá-los através de código.

Este texto não irá suprir os conhecimentos de geometria, geometria plana, geometrias espacial, geometria linear, ... que muitas vezes são necessárias para resolver os problemas, mas sim trazer algumas soluções computacionais para problemas comuns.

Pontos

O arquivo `pontos.cpp` traz a parametrização e rotinas básicas para tratar pontos no plano.

A constante `EPS` tem o valor de 10^{-9} e serve para representar quando não há diferença entre dois *doubles*, ao compará-los. Se a diferença entre eles for menor que `EPS` então ambos valores *double* são iguais, somente por questão de diferenças de arredondamento ao trabalhar com pontos flutuantes.

A classe `ponto_i` trabalha com um ponto no plano (coordenadas X e Y) representado por inteiros, em algumas situações é suficiente e mais eficiente. A classe `ponto_d` trabalha com ponto representado por *double*.

Todas as demais funções trabalha com o `template<class ponto>` que serve tanto para um quanto para outro.

Para os pontos, definimos a comparação de ordem, primeiro nas abscissas e, se as abscissas forem a mesma (dentro do arredondamento) então ordena pelas ordenadas, esta comparação de ordem é importante em problemas nos quais os pontos são ordenados, e podemos aplicar o `sort` em um vetor de pontos (Envoltória Convexa, por exemplo).

A partir da inclusão do `<utility>` toda e qualquer comparação entre pontos é possível.

As funções que definimos para pontos são:

Cálculo da distância entre dois pontos:
`double dist(const ponto& a, const ponto& b)`

$d = \sqrt{(b.x - a.x)^2 + (b.y - a.y)^2}$

Girar um ponto em sentido anti-horário de um ângulo *alfa*
`ponto girar(const ponto& p, double alfa)`

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

O ângulo entre três pontos $A\hat{O}B$ através de seu seno.

```
template<class ponto>
double angulo(const ponto& a, const ponto& o, const ponto& b);
```

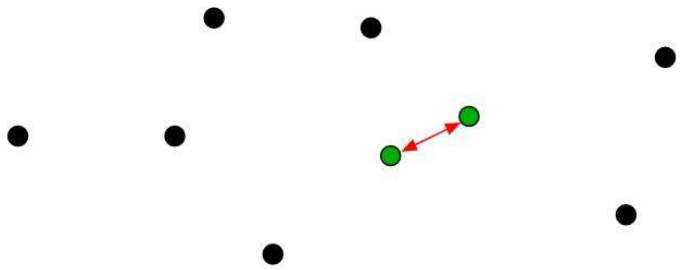
Considerando dois vetores: $\overrightarrow{OA}$ e $\overrightarrow{OB}$ $\sin arc = \dfrac{\overrightarrow{OA}_y \overrightarrow{OB}_x - \overrightarrow{OA_x'} \overrightarrow{OB_y'}}{\sqrt{|\overrightarrow{OA}||\overrightarrow{OB}|}}$

Muitas vezes queremos apenas o sinal do seno, que representa ângulo entre 0 e 180 (positivo) ou entre 180 e 360 (negativo), um ângulo zero representa pontos alinhados. Este sinal serve para a envoltoria convexa, lateralidade de um ponto em relação a um segmento, e outros. Quando for assim, dá para trabalhar com inteiros e esquecer a divisão pela raiz.

O problema do Par de Pontos mais próximos

Baseado nos slides de aulas da Prof. Dra. Cristina G. Fernandes

O problema: Dados n pontos no plano, determinar dois deles que estão à distância mínima.



A entrada do problema é determinada por uma coleção de pontos representadas por seus valores x e y, sendo que a distância entre dois pontos i e j é dada por: $d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

A saída esperada é a menor distância entre dois determinados pontos da coleção.

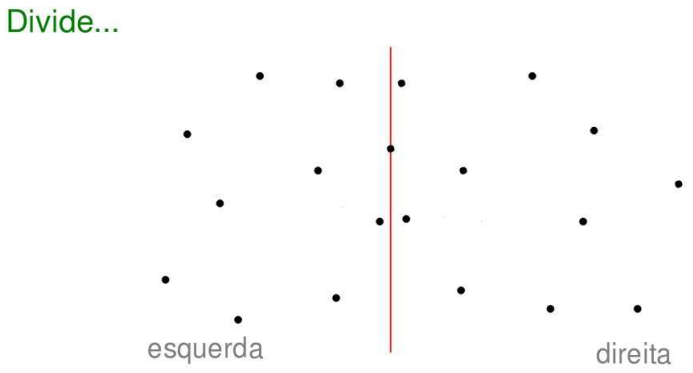
Força bruta: algoritmo quadrático, que testa todos os pares de pontos, esta implementação podemos ver no Algoritmo 1

Algoritmo 1 Algoritmo de força bruta para os pares mais próximos

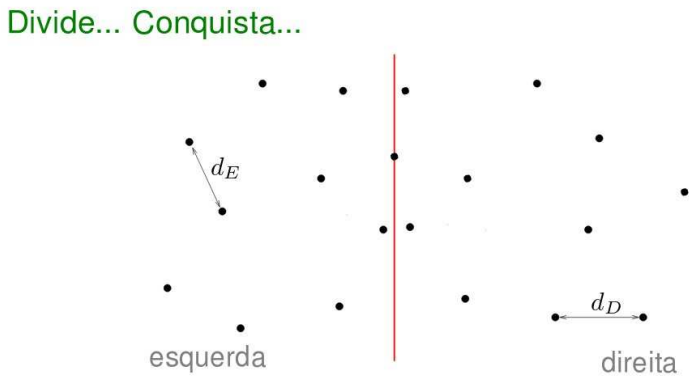
```
Algoritmo SIMPLES(P,n)
    d ← ∞
    para i ← 2 até n faça
        para j ← 1 até i − 1 faça
            se dist(P[i], P[j]) < d então
                d ← dist(P[i], P[j])
    Retorna d
```

Uma solução mais eficiente é o uso de divisão e conquista:

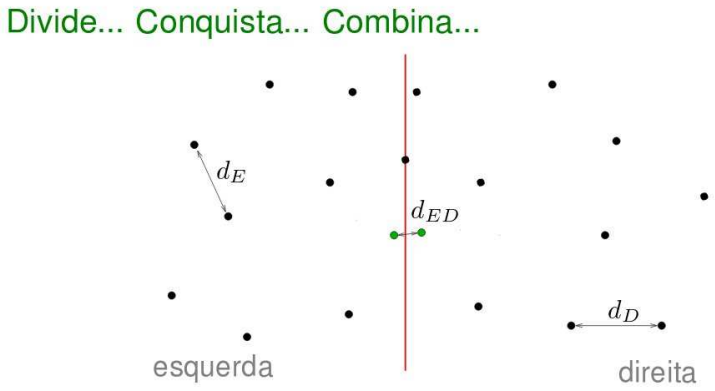
Para tanto, vamos dividir o plano em duas partes: Esquerda (E) e Direita (D): Divisão



Em cada lado podemos encontrar um par mais próximo: Conquista



Por fim, combinamos ambos os lados, nos preocupando de ver se na divisa existe um par com distância menor que aquele que encontramos na esquerda ou direita: Combine



Um exemplo de solução é o algoritmo de Shamos e Hoey. Inicialmente ordenamos os pontos pela abscissa (coordenada x), então aplicamos o algoritmo, veja na listagem do algoritmo 2, onde a chamada $DistanciaRecSH(P, 1, n)$ é a chamada recursiva do algoritmo de Shamos e Hoey no intervalo de pontos $[1..n]$.

Algoritmo 2 Aplicando o algoritmo de Shamos e Hoey

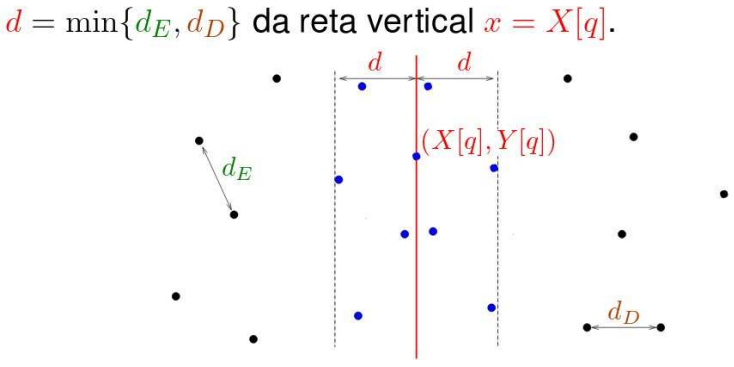
Algoritmo DistanciaSH(P,n)
  $Sort(P)$
  Retorna $DistanciaRecSH(P, 1, n)$

Como toda recursão, falamos de Indução Matemática. A base, e representado por 2 ou 3 pontos. Veja o algoritmo recursivo na listagem 3

Algoritmo 3 Versão Recursiva do Algoritmos de Shamos e Hoey

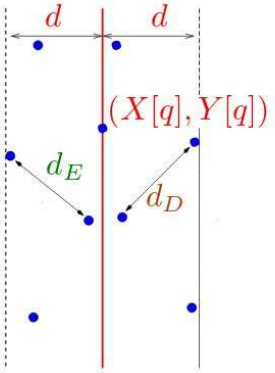
Algoritmo DistanciaSH(P,i,j)
  se $j \leqslant i + 2$ **então** ▷ Base
    $d \leftarrow$ ▷ Resolver o problema diretamente
  senão
    $k \leftarrow (i + j)/2$ ▷ Divisão
    $d_E \leftarrow DistanciaRecSH(P, i, q)$ ▷ Conquista
    $d_D \leftarrow DistanciaRecSH(P, q + 1, j)$
    $d \leftarrow \min\{d_E, d_D\}$
    $d \leftarrow Combine(P, i, j, k, d)$ ▷ Combinação
  Retorna d

Se conseguirmos fazer o *Combine* linear, teremos um algoritmo $O(n \log n)$. Como resolver então o *Combine*? Uma ideia é pegar os pontos que estão no máximo a uma distância $d = \min\{d_E, d_D\}$.

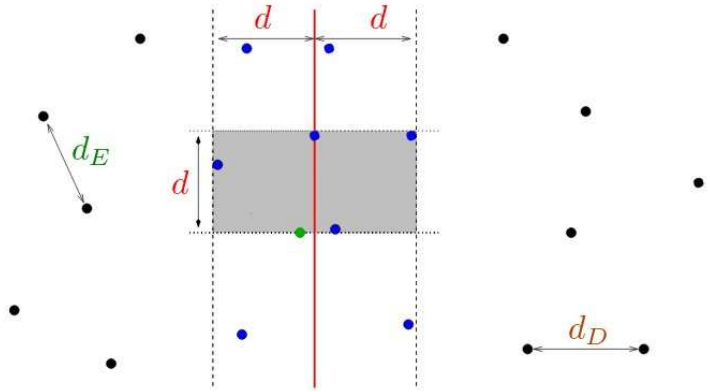


Esta estratégia não é suficiente, pois podemos cair no caso em que todos os pontos estão neste intervalo, logo o Combine volta a ser quadrático.

$d = \min\{d_E, d_D\}$ da reta vertical $x = X[q]$.



Podemos refinar nossa ideia, limitando também a distância no eixo das ordenadas (y)



Ou seja, para cada ponto na faixa, olhamos apenas para pontos da faixa que tenham coordenadas y no máximo a uma distância d acima do ponto. Quantos pontos seriam? Em cada um dos dois quadrados de lado d há no máximo 4 pontos, que podem estar nas laterais dos quadrados (se houver 2 pontos em uma lateral, então d seria menor). Juntanto os dois quadrados (uma mesma lateral) teremos no máximo 7 pontos neste intervalo. Precisamos identificar como alcançar estes pontos.

Podemos algerar o pré-processamento (ordenação) dos pontos, para ordená-los, também pela coordenada y, assim teríamos a sequência de pontos P ordenados pela abscissa (x) e a sequência de índices A indicando os índices dos pontos P ordenados pela ordenada y.

Vamos reescrever nosso algoritmo original na listagem 4:

Algoritmo 4 Aplicando o algoritmo de Shamos e Hoey

```
Algoritmo DistanciaSH(P,n)
    Sort(P)                                   ▷ Ordena P de acordo com P.x
    para t ← 1 até n faça
        A[i] ← i
    Sort(P, A)                                ▷ Ordena A de acordo com P[A].y
    Retorna DistanciaRecSH(P, A, 1, n)
```

Agora precisamos reescrever a versão recursiva de forma que a divisão leve em conta a nova ordenação. A divisão agora nos dará um vetor B de índices onde $B[i..k]$ é uma representação ordenada (seguindo a ordem de A) dos índices dos pontos $P[i..k]$ mais à esquerda da divisão, e $B[k+1..j]$ é uma representação ordenada dos índices dos pontos $P[k+1..j]$ mais à direita.

Veja o algoritmo recursivo na nova listagem 5

Se ambos, *Divida* e *Combine* forem lineares, então continuamos com $O(n \log n)$. Vamos primeiro ao *Divida*, que temos na listagem 6

Agora, para rodar o Combine, do algoritmo 5, precisamos identificar quais são os pontos que são candidatos ao cálculo da distância, ou seja que estão numa faixa de distância d em torno do ponto k, pois os pontos estão ordenados em y. Os candidatos estão no algoritmo 7

Algoritmo 5 Versão Recursiva do Algoritmos de Shamos e Hoey

Algoritmo DistanciaSH(P,A,i,j)
  se $j \leqslant i + 2$ **então** ▷ Base
    $d \leftarrow$ ▷ Resolver o problema diretamente
  senão
    $k \leftarrow (i + j)/2$ ▷ Divisão
    $B \leftarrow Divida(A, k, i, j)$
    $d_E \leftarrow DistanciaRecSH(P, B, i, k)$ ▷ Conquista
    $d_D \leftarrow DistanciaRecSH(P, B, k + 1, j)$
    $d \leftarrow \min\{d_E, d_D\}$
    $d \leftarrow Combine(P, A, i, j, k, d)$ ▷ Combinação
  Retorna d

Algoritmo 6 Divida, um algoritmo linear

Algoritmo Divida(A,k,i,j)
  $B \leftarrow A$
  $t \leftarrow i - 1$
  $r \leftarrow k$
  para $q \leftarrow i$ **até** j **faça**
    se $A[q] \leqslant k$ **então** ▷ Se refere a um ponto à esquerda (ordenado em y).
      $t \leftarrow t + 1$
      $B[t] \leftarrow A[q]$
    senão
      $r \leftarrow r + 1$
      $B[r] \leftarrow A[q]$
  Retorna B

Algoritmo 7 Candidatos: os pontos que em x estão a uma distância d do ponto k

Algoritmo Candidatos(P,A,i,j,k,d)
  $v \leftarrow 0$
  para $q \leftarrow i$ **até** j **faça**
    se $|P[A[q]].x - P[k].x| < d$ **então**
      $v \leftarrow v + 1$
      $F[v] \leftarrow A[q]$
  Retorna F, v

Finalmente podemos fazer o algoritmo *Combine*, pois já sabemos os candidatos e dentre os candidatos, no máximo os 7 pontos a partir de um determinado ponto poderia estar dentro da distância desejada. Veja o *Combine* na listagem 8

Algoritmo 8 Combine: para cada ponto na faixa, verifica se um dos próximos 7 têm distância menor

```
Algoritmo COMBINE(P,A,i,j,k,d)
    (F, v) ← Candidatos(P, A, i, j, k, d)        ▷ separamos só os pontos na faixa horizontal
    para t ← 1 até v − 1 faça
        para r ← i + 1 até min{i + 7, v} faça
            d' ← dist(P[F[t]], P[F[r]])
            se d' < d então d ← d'
    Retorna d
```

Neste algoritmo, se *Candidatos* trouxer todos os pontos, teremos no pior caso $T(n) = 7n$ pois para cada ponto temos um for de tamanho no máximo 7, e isto é $\Theta(n)$, logo, o *Combine* é linear.

Podemos melhorar um pouquinho para não precisar olhar os 7 próximos, pois podemos dispensar quem estiver com a distância y maior que d, veja a versão no algoritmo 9

Algoritmo 9 Combine: para cada ponto na faixa, verifica se um dos próximos 7 têm distância menor

```
Algoritmo COMBINE(P,A,i,j,k,d)
    (F, v) ← Candidatos(P, A, i, j, k, d)        ▷ separamos só os pontos na faixa horizontal
    para t ← 1 até v − 1 faça
        r ← i + 1
        enquanto (r ⩽ v) e (P[F[r]].y − P[F[t]].y < d) faça        ▷ ⩽ 7 próximos.
            d' ← dist(P[F[t]], P[F[r]])
            se d' < d então d ← d'
            r ← r + 1
    Retorna d
```

Este algoritmo pode ser ligeiramente modificado para retornar, ao invés da menor distância d entre os pontos, o par em si.

Vamos treinar:

URI - 1295

Neste problema é dado como entrada vários casos: sendo cada caso começando com N o número de pontos, e nas N linhas seguintes os N pontos descritos pelas coordenadas x e y, terminando com $N = 0$. A saída é `INFINITY` se a distância for maior que 10^4, ou se só houver um único ponto na entrada.

Escopo de uma solução que pode ser empregada está abaixo, fica como brincadeira implementar as funções, aqui em especial destacamos as funções de ordenação (com base no `sort` do `algorithm`) onde criamos uma classe (`ordemY`) com a implementação do `operator()` para ordenar em função de y, a ordenação em função de x se dá pela definição do `operator<` entre dois pontos.

Outro detalhe é que na função recursiva, ao criar o vetor B no `Divida()` o vetor B tem de ter o tamanho do vetor A para garantir que os índices sejam consistentes. `#define EPS 0.000000001`

```
class ponto { // Versão double
public:
 double x, y;
 ponto():  x(0.0), y(0.0) { }
 ponto(double a, double b):  x(a), y(b) { }
};

bool operator<(const ponto& a, const ponto& b) {
 if(fabs(a.x - b.x) < EPS)
  return a.y < b.y;
 else return a.x < b.x;
}

double dist(const ponto& a, const ponto& b) {
 double dx = b.x - a.x, dy = b.y - a.y;
 return sqrt(dx*dx + dy*dy);
}
```

```
class ordemY {
public:

 vector<ponto> P;

 ordemY(vector<ponto> V): P(V) { }
 bool operator()(int a, int b) {
  return (P[a].y < P[b].y);
 }
};

vector<int> Candidatos(const vector<ponto> &P, const vector<int> &A, int i, int j, int k, double d);
double Combine(const vector<ponto> &P, const vector<int> &A, int i, int j, int k, double d);
vector<int> Divida(const vector<int> &A, int i, int j, int k) {
 vector<int> B(A);
 ...
 return B;
}
double DistanciaRecSH(const vector<ponto> &P,const vector<int> &A, int i, int j);

int main() {
 int n;

 cin >> n;
 while(n) {
  double d;
  vector<ponto> P(n);
  vector<int> A(n);
  for(int i=0; i<n; i++) {
   double x, y;
   cin >> x >> y;
   P[i] = ponto(x,y);
   A[i] = i;
  }
  if(n<2) cout << "INFINITY<< endl;
  else {
   if(n < 4) {
    d = dist(P[0],P[1]);
    if(n==3) {
     double d1 = dist(P[0],P[2]);
     if(d > d1) d = d1;
     d1 = dist(P[1],P[2]);
     if(d > d1) d = d1;
    }
   } else {
    sort(P.begin(),P.end()); // Sort(P)
    sort(A.begin(),A.end(),ordemY(P)); // Sort(P,A)
    d = DistanciaRecSH(P,A,0,n-1);
   }
   if(d >= 10000) cout << "INFINITY<< endl;
   else cout << fixed << setprecision(4) << d << endl;
  }
  cin >> n;
 }
 return 0;
}
```