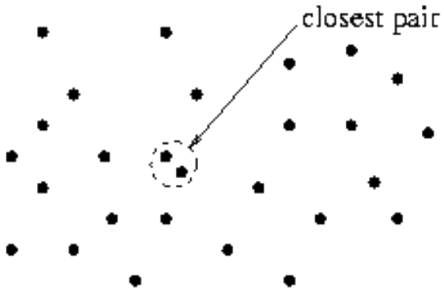
GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: PUNTOS MAS CERCANOS

1. Introducción

El problema de los puntos más cercanos (*Closest Pair Problem*) parte de un conjunto de *n* puntos pertenecientes al plano XY, donde cada punto está representado por sus coordenadas *X* y *Y*, se desea hallar encontrar el par de puntos tal que la distancia entre ambos puntos es mínima. El algoritmo debe devolver dichos puntos o la distancia .



2. Conocimientos previos

2.1. Distancia eucludiana

En matemáticas, la distancia euclidiana o euclídea, es la distancia .ºrdinaria.enstre dos puntos de un espacio euclídeo, la cual se deduce a partir del teorema de Pitágoras.

Por ejemplo, en un espacio bidimensional, la distancia euclidiana entre dos puntos P_1 y P_2, de coordenadas cartesianas (x_1, y_1) y (x_2, y_2) respectivamente, es:

$$d_e(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

2.2. Estrategia algoritmica: Divide y Conquista

Los algoritmos de este tipo se caracterizan por estar diseñados siguiendo estrictamente las siguientes fases:

- **Dividir:** Se divide el problema en partes más pequeñas.
- **Conquistar:** Se resuelven recursivamente los problemas mas chicos.
- **Combinar:** Los problemas mas chicos se combinan para resolver el grande.

Los algoritmos que utilizan este principio son netamente recursivos.

2.3. Estrategia algoritmica: Línea de barrido *Sweep Line*

En la geometría computacional, un algoritmo de línea de barrido o barrido de plano es un paradigma algorítmico que utiliza una línea de barrido conceptual o una superficie de barrido para

resolver diversos problemas en el espacio euclidiano . Es una de las técnicas clave en geometría computacional.

La idea detrás de los algoritmos de este tipo es imaginar que una línea (a menudo una línea vertical) se barre o se mueve a través del plano, deteniéndose en algunos puntos. Las operaciones geométricas están restringidas a objetos geométricos que se cruzan o se encuentran en las inmediaciones de la línea de barrido cada vez que se detiene, y la solución completa está disponible una vez que la línea ha pasado por todos los objetos.

3. Desarrollo

3.1. Solución iterativa

La solución iterativa a este problema es simple, se recorre la colección de puntos y por cada punto se le compara contra todos los demás calculando la distancia como $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$, cada vez que una distancia sea inferior al mínimo anterior actualizamos cuales son los puntos más cercanos. Cuando ya comparamos todos los puntos contra todos el resultado es el par de puntos cuya distancia es seguro es mínima.

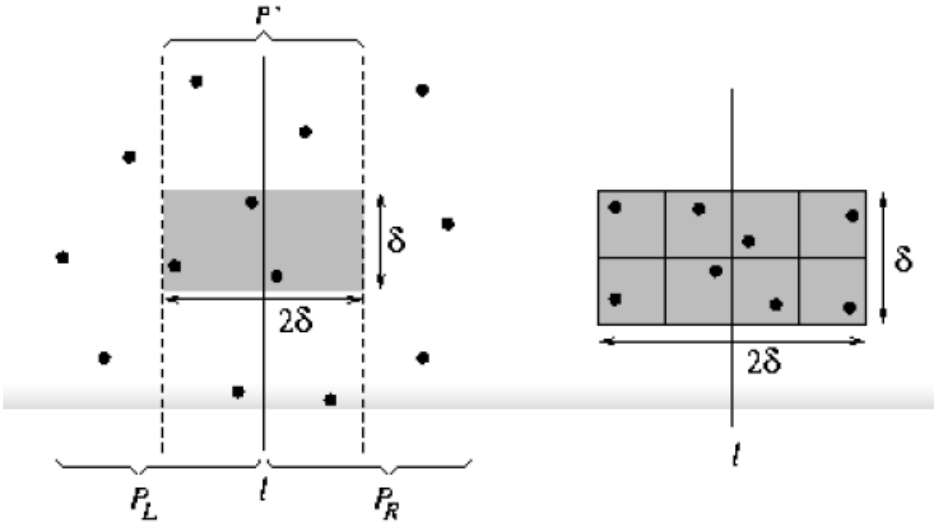
Es evidente que esta solución tanto en peor como en el mejor de los casos compara todos los puntos contra todos por lo que podemos decir que la complejidad de este algoritmo es O(n^2) siendo n la cantidad de puntos en el plano. Por lo que solo es recomendable usar esta variante en problemas de concursos cuando la cantidad de puntos no superen los 1000.

3.2. Divide y Conquista

- **Dividir:** Partimos que tenemos los puntos almacenados en alguna estructura de dato secuencial. Si son pocos puntos podemos aplicar la primera variante. Si hay más puntos de lo permisible entonces trazamos una línea vertical l que subdivide a la colección P de puntos en dos colecciones aproximadamente del mismo tamaño, *Pl* y *Pr*.
- **Conquistar:** Recursivamente aplicamos el procedimiento en ambas colecciones por lo que obtenemos δ_l y δr distancias mínimas para ambas colecciones. De ambas obtenemos $\delta = min(\delta_l, \delta r)$.
- **Combinar:** Lamentablemente δ no es el resultado final, ya que se podría darse la línea l que hemos elegido pasa justo entre dos puntos que están a distancia mínima. Debemos chequear si no hay dos puntos que están a distancia mínima. Debemos chequear si no hay dos puntos, uno a cada lado de la línea, cuya distancia sea menor que δ. En primer lugar observemos que no necesitamos chequear todos los puntos, si un punto esta a mayor distancia de l que δ entonces es seguro no hay vecino del otro lado de la línea que pueda formar una distancia menor que δ. Creamos una lista P_k de puntos que están a menos de δ de cada lado de l. Determinamos entonces la distancia mínima para los puntos de P_k que llamaremos δ' y devolvemos la distancia sino los dos puntos que están a dicha distancia uno del otro.

El problema consiste en analizar cómo encontrar la distancia mínima en P_k, lo cual requiere

un poco de astucia. Para optimizar el algoritmo queremos encontrar la distancia mínima en P_k en tiempo O(n) vamos demostrar que esto puede hacerse suponiendo que los puntos de P_k están ordenados por la coordenada y, luego tendremos que resolver como ordenar los puntos.



Supongamos que los puntos de P_k están ordenados de acuerdo a su coordenada y. Consideremos un punto P_k[i] en lista ordenada. Cuál es el punto más cercano a P_k[i]. Podemos restringir la búsqueda a los puntos que tienen índice j mayor que i ya que el proceso lo vamos a hacer para todos los puntos (si el vecino más cercano está arriba, entonces esto ya fue analizado cuando buscamos el vecino de dicho punto). Podríamos pensar que necesariamente el vecino más cercano es P_k[i+1], pero esto no es verdad, dos puntos que tienen distancia mínima en cuanto a la coordenada y no necesariamente tienen distancia mínima en el plano, queremos saber que tan lejos tenemos que buscar hasta P_k[i+2]?, P_k[i+8]? tal vez podamos acotar la búsqueda a una cantidad constante de puntos.

Resulta que podemos limitar la cantidad de puntos a buscar y además resulta que no nos hace falta analizar más de 7 puntos en la lista P_k para cada punto.

Resumiendo el funcionamiento del algoritmo:

- **Presort:** Dada la lista P, hacemos dos copias PX y PY. Ordenamos PX por la coordenada x y la lista PY por las coordenadas y.
- **Parte Recursiva:** DistMin(X,Y)
  - bf Condición de corte: Si la cantidad de puntos es menor que 4 entonces resolvemos el problema por fuerza bruta y devolvernos la distancia mínima δ analizando todas las distancias posibles.
  - **Dividir:** Sino, sea l la mediana de las coordenadas x de PX. Dividimos ambas listas PX y PY por esa línea, manteniendo el orden, creando X_l, X_r, Y_l, Y_r .
  - **Conquistar:** $\delta_l = DistMin(X_l, Y_l)$, $\delta_r = DistMin(X_r, Y_r)$.

- **Combinar:** $\delta = min(\delta_l, \delta r)$. Creamos la lista Y_l copiando todos los puntos de Y que están a distancia menor que δ de l. Para i desde 1 hasta la longitud de Y_l y para j desde i+1 hasta i+7 calcular la distancia entre Y_l[i] y Y_l[j]. Sea δ' la distancia mínima entre dichas distancias. Devolver min(δ, δ').

3.3. Línea de barrido

Barreremos el plano de izquierda a derecha (o de derecha a izquierda es una posibilidad) y cuándo se alcance alcance un punto computaremos todos los puntos candidatos cercanos a este (los candidatos que puede estar en el par más cercano).

Por lo que haremos las siguientes operaciones:

1. Ordenamos los puntos de izquierda a derecha por el eje x.
2. Por cada punto:
   a) Quitamos de los candidatos todo el punto que está más allá en el eje x de la distancia mínima actual.
   b) Tomamos a todos los candidatos que son localizados a igual o menor distancia que la distancia mínima del punto actual en eje vertical.
   c) Probamos para la distancia mínima todos los candidatos encontrados con el punto actual.
   d) Y finalmente le añadimos el punto actual a la lista de candidatos.

Así es que cuando encontramos una distancia mínima nueva , podemos hacer más pequeños el rectángulo de candidatos en el eje de las abscisas y más pequeño en el eje vertical. Así hacemos mucho menos comparaciones entre los puntos.

4. Implementación

4.1. C++

4.1.1. Solución iterativa

```
#include <float.h>
#include <stdlib.h>
#include <math.h>
#include <algorithm>

struct Point {
   double X,Y;
};

double dist(Point p1, Point p2){
   return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +(p1.y - p2.y)*(p1.y - p2.y));
```

```
}

double closestPair(Point P[], int n){
   double minDistance = FLT_MAX;
   for (int i = 0; i < n; ++i)
      for (int j = i+1; j < n; ++j)
         if (dist(P[i], P[j]) < minDistance)
            minDistance = dist(P[i], P[j]);
   return minDistance;
}
```

4.1.2. Divide y Conquista

```
#include <float.h>
#include <stdlib.h>
#include <math.h>
#include <algorithm>

struct Point {
   double X,Y;
};

int compareX(const void* a, const void* b){
   Point *p1 = (Point *)a, *p2 = (Point *)b;
   return (p1->X - p2->X);
}

int compareY(const void* a, const void* b){
   Point *p1 = (Point *)a,*p2 = (Point *)b;
   return (p1->Y - p2->Y);
}

double dist(Point p1, Point p2){
   return sqrt( (p1.X - p2.X)*(p1.X - p2.X) +(p1.Y - p2.Y)*(p1.Y - p2.Y));
}

double closestPair(Point P[], int n){
   double minDistance = FLT_MAX;
   for (int i = 0; i < n; ++i)
      for (int j = i+1; j < n; ++j)
         if (dist(P[i], P[j]) < minDistance)
            minDistance = dist(P[i], P[j]);
   return minDistance;
}

double stripClosest(Point strip[], int size, double d){
   double minDistance = d;
   qsort(strip, size, sizeof(Point), compareY);
```

```
   for (int i = 0; i < size; ++i)
      for (int j = i+1; j < size && (strip[j].Y - strip[i].Y) < minDistance;
          ++j)
         if (dist(strip[i],strip[j]) < minDistance)
            minDistance = dist(strip[i], strip[j]);
    return minDistance;
}

double closestUtil(Point P[], int n){
   if(n <= 3)
     return closestPairV1(P, n);
   int mid = n/2;
   Point midPoint = P[mid];
   double dl = closestUtil(P, mid);
   double dr = closestUtil(P + mid, n-mid);

   double d = min(dl, dr); Point strip[n]; int j = 0;
   for (int i = 0; i < n; i++)
      if (abs(P[i].X - midPoint.X) < d){
        strip[j] = P[i];
        j++;
      }
   return min(d, stripClosest(strip, j, d) );
}

double closestPairV2(Point P[], int n){
   qsort(P, n, sizeof(Point), compareX);
   return closestUtil(P, n);
}
```

4.1.3. Línea de barrido

```
#include <float.h>
#include <stdlib.h>
#include <math.h>
#include <algorithm>

struct Point {
   double X,Y;
   Point (double _X=0,double _Y=0){
      X=_X; Y=_Y;
   }

   bool operator<(const Point &P) const{
      if(X < P.X) return true;
      if(X > P.X) return false;
      return Y < P.Y;
   }
```

```
};

double distancePointToPoint(Point _a,Point _b){
   double dist=(_a.X-_b.X)*(_a.X-_b.X)+(_a.Y-_b.Y)*(_a.Y-_b.Y);
   return sqrt(dist);
}

/*Los puntos deben ser almacenados a partir de la posicion 1 en adelante y no
   en la posicion 0 como de costumbre*/
double closestPair(Point _points [], int _npoint){
   sort(_points+1,_points+_npoint+1);
   if(_points == 0) return 0;
   /* INF debe ser un valor que sea mayor que la distancia maxima en que se
      pueda encontrar dos puntos segun el problema. Si dicen que las
      coordenadas de los puntos va estar entre -10^9 a 10^9 entonces: INF=
      distancePointToPoint( Point(-10^9,-10^9),Point(10^9,10^9) )+100 */
   double dist = INF;
   int tBegin = 1, tEnd = 1, tot = 1;
   for(int i=2;i<=_npoint;i++){
      while(tot > 0 && 0.0+_points[i].X-_points[tBegin].X > dist){
          tBegin++; tot--;
      }
      for(int j=tBegin;j<=tEnd;j++)
          dist = min(dist, distancePointToPoint(_points[i], _points[j]));
      tEnd++; tot++;
   }
   return dist;
}
```

4.2. Java

4.2.1. Solución iterativa

```
private class Point{
   public double x;
   public double y;

   public Point(int _x,int _y) {
      this.x =_x; this.y =_y;
   }
}

private double closestPair(Point P[], int n){
   double minDistance = Double.MAX_VALUE;
   for (int i = 0; i < n; ++i)
      for (int j = i+1; j < n; ++j)
         minDistance = Math.min(minDistance,dist(P[i], P[j]));
   return minDistance;
```

```
}

private double dist(Point p1, Point p2) {
   return Math.hypot(p1.x-p2.x, p1.y-p2.y);
}
```

4.2.2. Divide y Conquista

```
class Point {
   public int x;
   public int y;

   Point(int x, int y) {
      this.x = x;this.y = y;
   }

   public static float dist(Point p1, Point p2){
      return (float) Math.sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) *
          (p1.y - p2.y));
   }

   public static float bruteForce(Point[] P, int n) {
      float minD = Float.MAX_VALUE;float currMin = 0;
      for (int i = 0; i < n; ++i) {
         for (int j = i + 1; j < n; ++j) {
            minD = Math.min(minD, dist(P[i], P[j]));
         }
      }
      return min;
   }

   public static float stripClosest(Point[] strip, int size, float d) {
      float minD = d;
      Arrays.sort(strip, 0, size, new PointYComparator());
      for (int i = 0; i < size; ++i) {
         for (int j = i + 1; j < size && (strip[j].y - strip[i].y) < min; ++j)
             {
            minD = Math.min(minD, dist(strip[i], strip[j]));
         }
      }
      return min;
   }

   public static float closestUtil(Point[] P,int startIndex, int endIndex){
      if ((endIndex - startIndex) <= 3) { return bruteForce(P, endIndex); }
      int mid = startIndex + (endIndex - startIndex) / 2;
      Point midPoint = P[mid];
      float dl = closestUtil(P, startIndex, mid);
```

```
      float dr = closestUtil(P, mid, endIndex);
      float d = Math.min(dl, dr);

      Point[] strip = new Point[endIndex];
      int j = 0;
      for (int i = 0; i < endIndex; i++) {
         if (Math.abs(P[i].x - midPoint.x) < d) {
            strip[j] = P[i];j++;
         }
      }
      return Math.min(d, stripClosest(strip, j, d));
   }

   public static float closest(Point[] P, int n) {
      Arrays.sort(P, 0, n, new PointXComparator());
      return closestUtil(P, 0, n);
   }
}

class PointXComparator implements Comparator<Point> {
   @Override
   public int compare(Point pointA, Point pointB) {
      return Integer.compare(pointA.x, pointB.x);
   }
}

class PointYComparator implements Comparator<Point> {
   @Override
   public int compare(Point pointA, Point pointB) {
      return Integer.compare(pointA.y, pointB.y);
   }
}
```

4.2.3. Línea de barrido

```
private double distance(Point p1, Point p2) {
   return Math.sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}

/*Los puntos deben ser almacenados a partir de la posicion 1 en adelante y no
   en la posicion 0 como de costumbre*/
private double closestPair(Point [] _points, int _npoints) {
   Arrays.sort(_points,1,_npoints+1);
   if(_npoints!=0) {
      double dist = Double.MAX_VALUE;
      int tBegin = 1 , tEnd = 1 , tot = 1;
      for(int i=2;i<=_npoints;i++){
         while(tot > 0 && 0.0+_points[i].x-_points[tBegin].x > dist){
```

```
            tBegin++; tot--;
          }
          for(int j=tBegin;j<=tEnd;j++)
             dist = Math.min(dist,distance(_points[i], _points[j]));
          tEnd++;tot++;
        }
        return dist;
     }
     return 0;
}

private class Point implements Comparable<Point>{
   public double x;
   public double y;
   public Point(int _x,int _y) {
      this.x =_x;
      this.y =_y;
   }
   @Override
   public int compareTo(Point p1) {
      if(this.x < p1.x) return -1;
      else if (this.x > p1.x) return 1;
      else{
         if(this.y < p1.y) return -1;
         else if(this.y > p1.y) return 1;
         else return 0; }
   }
}
```

5. Complejidad

Es evidente que la solucion iterativa tanto en peor como en el mejor de los casos compara todos los puntos contra todos por lo que podemos decir que la complejidad de este algoritmo es O(n^2) siendo n la cantidad de puntos en el plano. Por lo que solo es recomendable usar esta variante en problemas de concursos cuando la cantidad de puntos no superen los 1000.

Debemos analizar cuanto tiempo tarda este algoritmo con la estrategia divide y conquista en su peor caso, en el caso promedio en análisis es muy complejo porque depende de cuantos puntos tiene P_k en promedio y esto depende en cual es la distancia mínima esperada en cada sublista. En el peor de los casos podemos suponer que P_k tiene a todos los puntos. La fase pre-sort insume O(nlogn) igual valor arroja la parte recursiva del algoritmo por lo que obtenemos un algoritmo O(nlogn) mas eficiente que el algoritmo de fuerza bruta que era O(n^2).

De forma similar el algoritmo que utiliza la línea de barrido como estrategia su complejidad es O(nlogn) y su implementación es mas corta que utilizando que el algoritmo con la estrategia divide y conquista.

6. Aplicaciones

Es evidente que la aplicación de este algoritmo permite hallar dado una colección de puntos las distancia o los puntos mas cercanos. A pesar que analizamos para puntos de dos dimensiones los algoritmos expuestos se pueden extrapolar para puntos de tres dimensiones.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se pueden resolver aplicando este algoritmo:

- DMOJ - Vacas Claustrofóbicas.
- Kattis - Closest Pair
- SPOJ - CPP - Closest Pair Problem
- UVA - 10245 - The Closest Pair Problem
- AIZU Online Judge - Closest Pair