CFA INSTA
Institut National Supérieur des Technologies Avancées

L'utilisation d’algorithme d’Emo Welzl pour resoudre le problème du cercle minimum

Table de matière

1. Introduction

Dans le domaine en constante évolution de la géométrie computationnelle, la recherche de solutions efficaces et optimisées pour des problèmes complexes est une quête perpétuelle. Parmi ces défis se trouve le problème du cercle minimum, une question qui, à première vue, semble simple, mais dont la complexité et les applications pratiques s'étendent bien au-delà des limites théoriques. C'est dans ce contexte que l'algorithme de Welzl, introduit par Emo Welzl en 1991, se révèle être un outil puissant et élégant, offrant une méthode efficace pour résoudre ce problème classique de géométrie.

Le problème du cercle minimum, qui consiste à trouver le plus petit cercle englobant un ensemble de points dans un plan, trouve ses applications dans de nombreux domaines tels que l'optimisation de réseaux, la planification spatiale en robotique, et le traitement de données spatiales. L'algorithme de Welzl, avec sa complexité temporelle impressionnante et son approche récursive, n'est pas seulement une prouesse mathématique, mais aussi une solution pratique dans des situations où l'efficacité et la précision sont cruciales.

Cette recherche vise à explorer en profondeur l'algorithme de Welzl, en commençant par une compréhension du problème du cercle minimum. Nous examinerons l'importance de cet algorithme dans le contexte actuel, en soulignant comment il continue d'influencer non seulement la géométrie computationnelle, mais aussi d'autres domaines où la résolution rapide et efficace de problèmes est essentielle. En outre, cette étude vise à démystifier les aspects techniques de l'algorithme, tout en rendant ses concepts accessibles à un public plus large, afin de mettre en lumière l'élégance et l'utilité de cette méthode dans le monde moderne.

2. Problème de cercle minimum

2.1. Définition

Pour un ensemble fermé borné du plan, il existe un unique cercle de rayon minimal englobant l'ensemble. Son centre est le point pour lequel la plus grande distance à un point de l'ensemble est la plus petite possible. Ce cercle minimum est en général dénommé cercle circonscrit, sauf dans le cas d'un triangle obtus où le cercle minimum est le cercle de diamètre le plus grand côté.

source wikipédia : Centre (géométrie) — Wikipédia (wikipedia.org)

2.2. Cas de base pour la détermination d'un cercle :

Cercle passant par un seul point (x) :

Ce cercle a pour centre le point x lui-même et un rayon nul.

Cercle passant par deux points (p et q) :

Centre du cercle (c) : Le point moyen des deux points p et q, calculé comme :

- c = point [(xp+xq)/2 , (yp+yq)/2]
- Rayon (r) : La distance euclidienne entre p et q, notée distance(p, q).

Cercle passant par trois points (a, b, c) :

Il s'agit du cercle circonscrit au triangle formé par les points a, b et c.

2.3. Cercle formé grâce à 3 points

Le cercle circonscrit au triangle formé par les points a, b et c est défini par les formules suivantes :

Calcul du dénominateur (d) :

d = 2 * (ax * (by - cy) + bx * (cy - ay) + cx * (ay - by))

Norme d'un point (Point a) :

norm(a) = ax^2 + ay^2

Coordonnées du centre (xc, yc) :

xc = (norm(a) * (by - cy) + norm(b) * (cy - ay) + norm(c) * (ay - by)) / d

yc = (norm(a) * (cx - bx) + norm(b) * (ax - cx) + norm(c) * (bx - ax)) / d

Définition du cercle circonscrit :

Centre du cercle : c = point[xc, yc]

Rayon du cercle : r = distance(a, c)

3. Algorithme naïf

3.1. Détermination du Cercle Minimum Contenant un ensemble de Points

Considérons P, un ensemble de points dans un plan. Soit C le cercle de rayon minimum r qui englobe tous les points de P. La détermination de ce cercle s'appuie sur les deux lemmes suivants :

Lemme 1 (Cercle couvrant de diamètre minimal) :

Si un cercle, dont le diamètre est égal à la distance entre deux points quelconques de P, englobe tous les autres points de P, alors ce cercle est le cercle couvrant de rayon minimum pour l'ensemble P.

Lemme 2 (Unicité du cercle en 2D) :

Dans un plan bidimensionnel (2D), il existe un unique cercle qui passe par trois points non alignés (non-colinéaires).

3.2. Algorithme naïf

Cet algorithme, basé sur les deux lemmes précédentes, permet de trouver le cercle minimum c pour un ensemble donné de points, nommé "points". Son fonctionnement est le suivant :

Étape 1 : Examiner les Paires de Points :

- ➔ Parcourez chaque paire de points (p et q) dans l'ensemble "points".
- ➔ Pour chaque paire, vérifiez si le cercle défini par ces deux points englobe tous les autres points de l'ensemble.
- ➔ Si un tel cercle existe, il est retourné comme le cercle minimum c.

Étape 2 : Examiner les Trios de Points :

- ➔ Si aucun cercle couvrant minimal n'est trouvé avec les paires de points, procédez à l'examen des trios de points.
- ➔ Parcourez chaque trio de points (p, q et r) dans l'ensemble "points", en veillant à ce que les points ne soient pas colinéaires.
- ➔ Pour chaque trio, vérifiez si le cercle circonscrit au triangle formé par p, q et r englobe tous les points de l'ensemble.
- ➔ Cette méthode systématique permet d'identifier le cercle minimum couvrant l'ensemble de points en utilisant une approche étape par étape.

Pseudo-code :

pour tout p dans Points
pour tout q dans Points
c ← cercle de centre (p+q)/2 de diamètre |pq| 2
si c couvre tous les points de Points alors retourner c
résultat ← cercle de rayon infini

pour tout p dans Points
pour tout q dans Points
pour tout r dans Points
c ← cercle circonscrit de p, q et r
si c couvrePoints et c plus petit que résultat
alors resultat ← c
retourner c

3.3. Implémentation sous Java

Cette partie est longue, donc, je décide de les mettre dans l'annexe.

3.4. Niveau de complexité

Première Partie (Trois Boucle For) :

Complexité : O(n^3) - où n est le nombre de points.

Pour chaque paire de points (O(n^2)), il vérifie tous les autres points (O(n)).

Seconde Partie (Quatre Boucle For) :

Cette section examine chaque triplet de points (i, j, k) et calcule le cercle circonscrit si les points ne sont pas colinéaires.

Complexité : O(n^4) - pour chaque triplet de points (O(n^3)), il vérifie tous les autres points (O(n)) pour déterminer si le cercle est le plus petit cercle englobant.

La partie avec la complexité la plus élevée (O(n^4)) domine la performance globale du code.

Conclusion sur la complexité :

Cet algorithme naïf a une complexité temporelle globale de O(n^4), ce qui signifie qu'il devient rapidement inefficace à mesure que la taille de l'ensemble de points augmente.

Pour de grands ensembles de points, cet algorithme pourrait être très lent. Nous avons essayé de lancer cet algorithme avec l'input de 10000 points, et cette action rend la machine inactive.

Autre algorithme possible ?

Considérez des algorithmes plus efficaces pour résoudre ce problème, comme l'algorithme de Welzl, qui a une complexité moyenne en O(n) et est beaucoup plus efficace pour de grands ensembles de points.

Réduire le nombre de vérifications inutiles, par exemple, en pré-traitant les points pour éliminer les doublons ou en utilisant des structures de données plus efficaces pour les vérifications de collision.

4. Algorithme d'Emo Welzl

L'algorithme de Welzl est un algorithme récursif qui construit le plus petit cercle englobant à partir d'un ensemble de points. Il utilise une technique "Diviser pour régner" combinée à des propriétés géométriques.

4.1. Principe

Lemme 3

Soient P et R deux ensembles finis de points dans un plan, avec P non vide. Considérons un point p de P et désignons par *bmd(P,R)* le cercle minimum englobant.

Existence et Unicité :

Si un cercle peut contenir tous les points de P avec les points de R situés sur son bord, alors *bmd(P,R)* existe et est unique.

Inclusion de Point sur le Bord :

Si le point p n'est pas à l'intérieur de *b_md(P-{p},R)*, alors p se trouve sur le bord du cercle, à condition que ce dernier existe. Ainsi, *b_md(P,R)* est égal à *b_md(P-{p},R∪{p})*.

Sous-ensemble Déterminant :

Si *bmd(P,R)* existe, alors il y a un sous-ensemble S de P contenant au maximum *Max{0, 3-|R|} points* tels que *bmd(P,R)* soit équivalent à *bmd(S,R)*.

Cette lemme souligne l'efficacité de l'algorithme de Welzl, en mettant en évidence comment un petit sous-ensemble de points peut être utilisé pour déterminer le cercle minimum englobant, réduisant ainsi la complexité du problème.

4.2. Algorithme

Etape 1 : la fonction qui appel l'algorithme de Welzl

```
algorithmeWelzl(P un ensemble de points)
    Retourner b_minidisk(P, {})
Fin mindisk
```

Etape 2 : Algorithme de Welzl

```
b_minidisk(P un ensemble de points, R un ensemble de points)
    /* D non défini au début */
        D = φ
    /* On créé directement le cercle */
    Si (P est vide ou que |P| = 3)
        D = b_md({}, R)
    Fin Si
    Sinon
        /* Choix de p */
        Choisir un point p aléatoire de P
        /* Premier appel récursif */
        D = b_minidisk(P - {p}, R)
        Si (D est défini et que p n'appartient pas à D)
        /* Deuxième appel récursif */
        D = b_minidisk(P - {p}, R U {p})
        Fin Si
    Fin Sinon
    Retourner D
Fin b_minidisk
```

4.3. Implémentation sous Java

Cette partie est longue, donc, je décide de les mettre dans l’annexe.

4.4. Niveau de complexité

La Méthode Welzl et CercleMin

Ces méthodes sont au cœur de l'algorithme. L'algorithme de Welzl est un algorithme récursif qui utilise une stratégie de "Diviser pour régner" avec randomisation. En moyenne, sa complexité est en O(n), où n est le nombre de points.

La Méthode bmd

Cette méthode calcule le cercle minimum englobant pour des cas spécifiques où l'ensemble R contient 0, 1, 2, ou 3 points. Sa complexité est O(1) car elle ne dépend pas de la taille de l'ensemble des points P.

La Méthode cercle_avec_3point

Cette méthode est utilisée pour calculer un cercle à partir de trois points. La complexité de cette méthode est également O(1), car elle implique un nombre fixe d'opérations mathématiques indépendamment de la taille de l'ensemble des points.

La Méthode estInterieur

Cette méthode vérifie si un point est à l'intérieur d'un cercle. Sa complexité est O(1), car elle ne dépend pas de la taille de l'ensemble des points.

Conclusion sur la Complexité

En supposant une distribution moyenne des points et un fonctionnement optimal de la randomisation, la complexité globale de votre algorithme est **O(n)**. C'est une estimation moyenne, avec des cas exceptionnels où la complexité pourrait atteindre O(n^3) en raison de la nature récursive et de la randomisation.

5. Exécution et Résultat

5.1. Contexte

- Equipe de projet : M. DAM Thai Long & M. PHAM Quang Minh
- Machine d'exécution : Macbook pro avec CPU Apple Silicon M2 Max 32 Go RAM.
- Les scripts sont développés en Java avec l'outil IntelliJ IDEA 2023.3.1.
- Base de test : Varoumas_benchmark (*algo/samples*)
- Temps d'exécution:
    - en milliseconde : System.currentTimeMillis()
    - en nanoseconde : System.nanoTime()
    - On essaie d' échanger entre milliseconde et nanoseconde.

      Finalement, on décide d'utiliser nanoseconde puis diviser par 1000 pour trouver le résultat en microseconde pour la visualisation de données. Car en System.currentTimeMillis() le temps d'exécution est généralement 0 millisec.
- Traitement des données : Microsoft Excel.

5.2. Résultat d'exécution

Les résultats enregistrés dans le fichier csv puis transformer et visualiser sous forme de diagram par Excel. Vous pouvez consulter les résultats (en csv et exce) dans le dossier ***output***.

En passant un base de test de 1664 cas différent, ma machine s'arrête et le resultat contient seulement 808 cas de test accompagné par une lancement d'exception:

java.lang.NumberFormatException: For input string: "test-10

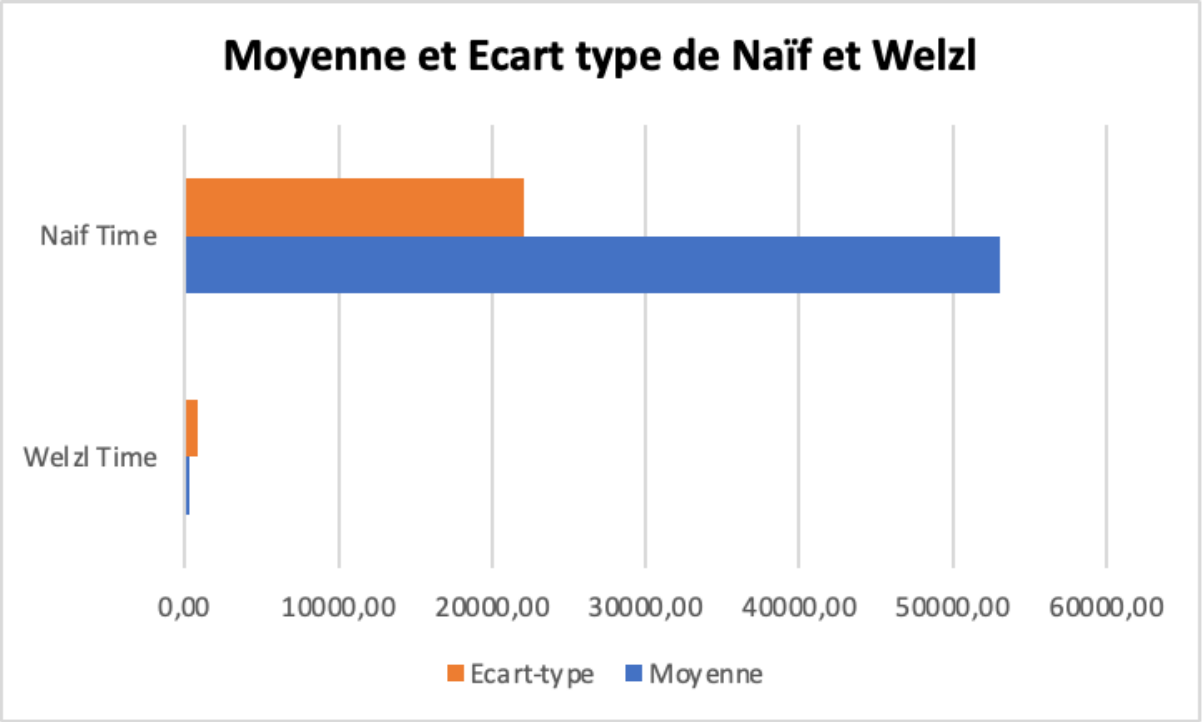
Ce problème d'exécution n'apparaît pas dans la machine de mon collègue.

Mais, avec 808, je crois que l'on a une dataset largement suffisant pour analyser.

Voici la moyenne et écart-type de ces deux solutions :

	Welzl Time	**Naif Time**
Moyenne	270,73	53063,44
Ecart-type	814,27	22053,81

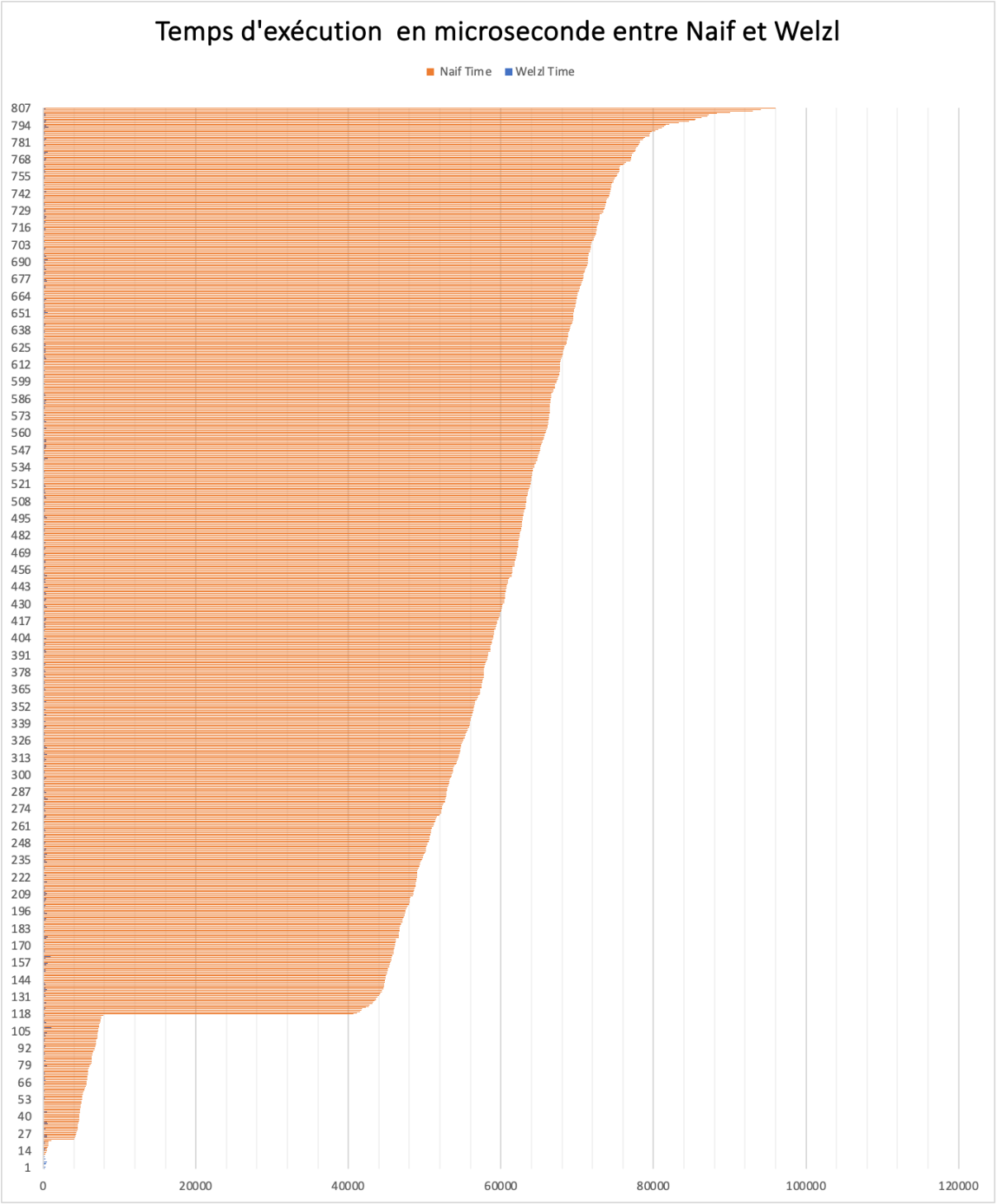
Tableau du temps d'exécution moyenne et l'écart-type entre deux solutions



En observant la courbe distribuée le résultat enregistré, après le tri des donnée par l'ordre croissance de temps d'exécution de l'algorithme naïf, je trouve qu'il y a une énorme écart de temps entre ces deux.

Pour Naïf, le temps est aller jusqu'à ~ 100 000 microsec, tandis que le temps max de Welzl est ~23 000 microsec (1 seul cas). En effet, le temps moyen de Welzl est très très bas, presque 0 en microsec, et celui de naïf est de 53 000 microsec.

Pour expliquer cette différence, on rappelle la complexité de ces deux algorithmes. Le grandeur de Welzl est O(n) et O(n^4) de naïf. En effet, dans naïf, on utilise quatre boucles imbriquées, ceci rend une perte de performance du machine. Plus on injecte les bases de test, plus le système est chargé. Puis, ceci rend une stackoverflow à la machine.


Temps d'exécution en microseconde entre Naif et Welzl
Naif Time
Welzl Time
807
794
781
768
755
742
729
716
703
690
677
664
651
638
625
612
599
586
573
560
547
534
521
508
495
482
469
456
443
430
417
404
391
378
365
352
339
326
313
300
287
274
261
248
235
222
209
196
183
170
157
144
131
118
105
92
79
66
53
40
27
14
1
0
20000
40000
60000
80000
100000
120000

Analyse des résultats des cercles produits par les deux algorithmes :

Pour calculer l'aire des cercles, nous utilisons le type double en Java, afin de garantir la plus grande précision possible. Il est observé que les cercles résultants des deux algorithmes présentent généralement des aires, et par conséquent des diamètres, différents. Cette différence s'explique par les méthodes distinctes employées par les deux algorithmes pour aborder la résolution du problème.

Étant donné que l'algorithme naïf est conçu pour fournir un résultat précis, nous pouvons en déduire que l'algorithme de Welzl n'offre pas systématiquement un résultat exact. Toutefois, dans la majorité des cas, l'algorithme de Welzl produit un cercle très similaire, voire identique, à celui obtenu avec l'algorithme naïf. Cette tendance souligne l'efficacité de l'algorithme de Welzl dans la pratique, malgré les différences de précision occasionnelles par rapport à l'approche naïve.

Conclusion

L'algorithme naïf sert de référence essentielle dans cette étude pour évaluer l'algorithme de Welzl. Il offre une compréhension simplifiée du problème du cercle minimum englobant. Cependant, son inconvénient majeur réside dans sa complexité en pire cas, qui est de l'ordre de O(n^4), résultant en des temps de calcul prohibitifs pour de grands ensembles de points.

D'autre part, l'algorithme proposé par Welzl est incrémental et repose sur l'idée que chaque point de l'ensemble est soit à l'intérieur du cercle, soit sur le périmètre du cercle minimum englobant tous les points. Sa complexité en pire cas est linéaire, c'est-à-dire en O(n).

Les tests effectués avec les données de Varoumas Benchmark montrent que l'algorithme de Welzl est performant en termes de temps de calcul, généralement de l'ordre de la milliseconde. Il est important de noter que, étant un algorithme récursif, il génère un grand nombre d'appels de fonctions, ce qui peut saturer la pile d'exécution et limiter son efficacité sur des ensembles de données très volumineux. Par conséquent, bien que performant, l'algorithme de Welzl doit être utilisé avec prudence sur de très grands ensembles de points pour éviter les problèmes de saturation de la pile d'appels de fonction.

Annexe

Implémentation sous java de l'algorithme de Welzl

```
/** circle de l'algorithme de  Welzl */

 public Circle Welzl(ArrayList<Point> points) {
   return CercleMin(points, new ArrayList<Point>());
 }

 private Circle CercleMin(ArrayList<Point> input, ArrayList<Point>
R ) {
   //P : liste des points donnée
   //R : liste des points sur la frontière et vide

   ArrayList<Point> P = new ArrayList<Point>(input);
   Random random = new Random();
   Circle cercle = null;

   if (P.isEmpty() || R.size() == 3) {
     cercle = bmd(new ArrayList<Point>(), R);

   } else {
     Point p = P.get((random.nextInt(P.size())));
     P.remove(p);

     //recursive : la liste P est modifié
     cercle = CercleMin(P, R);

     if (cercle != null && !estInterieur(cercle, p)) {
       R.add(p);
       cercle = CercleMin(P, R);
       R.remove(p);
     }
   }

   return cercle;
```

```
 }

 private boolean estInterieur(Circle c, Point p) {
   if (p.distance(c.getCenter()) - c.getRadius() < 0.00001) {
     return true;
   }
   return false;
 }

//dessiner le base minimum disk, un cercle avec le minimum de
point
 private Circle bmd(ArrayList<Point> P, ArrayList<Point> R) {
   if (P.isEmpty() && R.size() == 0)
     return new Circle(new Point(0, 0), 10);

   Random r = new Random();
   Circle cercle = null;

    //Si il y a un seul point dans la list, le cercle est ... ce
point
   if (R.size() == 1) {
     cercle = new Circle(R.get(0), 0);
   }

    //ily a 2 point, le cercle sera formé avec le centre étant le
milieu de la distance de ces 2 points
   if (R.size() == 2) {
     double cx = (R.get(0).x + R.get(1).x) / 2;
     double cy = (R.get(0).y + R.get(1).y) / 2;
     double d = R.get(0).distance(R.get(1)) / 2;
     Point p = new Point((int) cx, (int) cy);
     cercle = new Circle(p, (int) Math.ceil(d));
   } else {
     if (R.size() == 3)
        cercle = cercle_avec_3point(R.get(0), R.get(1), R.get(2));
//on utilise le méthode de faire la cercle avec 3 points
   }
```

```
    return cercle;
 }

//dessiner un cercle avec 3 point
 private Circle cercle_avec_3point(Point a, Point b, Point c) {
     // Calcul du déterminant pour vérifier si les points sont
colinéaires.
    double d = (a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y
- b.y)) * 2;

    // Si les points sont colinéaires, aucun cercle unique ne peut
être formé.
    if (d == 0) return null;

    double x = ((norm(a) * (b.y - c.y)) + (norm(b) * (c.y - a.y)) +
(norm(c) * (a.y - b.y))) / d;

    double y = ((norm(a) * (c.x - b.x)) + (norm(b) * (a.x - c.x)) +
(norm(c) * (b.x - a.x))) / d;

    Point p = new Point((int) x, (int) y);

    //constructeur du Circle a besoind un type de int
     // Math.ceil pour s'assurer que le rayon du cercle est assez
grand pour inclure le point a, même si la distance exacte est un
nombre non entier.
    return new Circle(p, (int) Math.ceil(p.distance(a)));
 }

//maths : pour calculer la valeur au carré de chaque coordonnée
 private int norm(Point a) {
    return (a.x * a.x) + (a.y * a.y);
 }
```

Implémentation sous java de l’algorithme naïf

```
public Circle naif(ArrayList<Point> inputPoints){

 ArrayList<Point> points = (ArrayList<Point>) inputPoints.clone();

 if (points.size()<1) return null;

 double cX,cY,cRadius,cRadiusSquared;

 for (Point p: points){
   for (Point q: points){
     cX = .5*(p.x+q.x);
     cY = .5*(p.y+q.y);
                                              cRadiusSquared          =
0.25*((p.x-q.x)*(p.x-q.x)+(p.y-q.y)*(p.y-q.y));
     boolean allHit = true;
     for (Point s: points)
       if ((s.x-cX)*(s.x-cX)+(s.y-cY)*(s.y-cY)>cRadiusSquared){
         allHit = false;
         break;
       }
                    if     (allHit)     return     new     Circle(new
Point((int)cX,(int)cY),(int)Math.sqrt(cRadiusSquared));
   }
 }

 double resX=0;
 double resY=0;
 double resRadiusSquared=Double.MAX_VALUE;
 for (int i=0;i<points.size();i++){
   for (int j=i+1;j<points.size();j++){
     for (int k=j+1;k<points.size();k++){
       Point p=points.get(i);
       Point q=points.get(j);
       Point r=points.get(k);
       //si les trois sont colineaires on passe
       if ((q.x-p.x)*(r.y-p.y)-(q.y-p.y)*(r.x-p.x)==0) continue;
```

```
        //si p et q sont sur la meme ligne, ou p et r sont sur la
meme ligne, on les echange
       if ((p.y==q.y)||(p.y==r.y)) {
         if (p.y==q.y){
             p=points.get(k); //ici on est certain que p n'est sur
la meme ligne de ni q ni r
               r=points.get(i); //parce que les trois points sont
non-colineaires
         } else {
             p=points.get(j); //ici on est certain que p n'est sur
la meme ligne de ni q ni r
               q=points.get(i); //parce que les trois points sont
non-colineaires
         }
       }
          //on cherche les coordonnees du cercle circonscrit du
triangle pqr
       //soit m=(p+q)/2 et n=(p+r)/2
       double mX=.5*(p.x+q.x);
       double mY=.5*(p.y+q.y);
       double nX=.5*(p.x+r.x);
       double nY=.5*(p.y+r.y);
        //soit y=alpha1*x+beta1 l'equation de la droite passant par
m et perpendiculaire a la droite (pq)
        //soit y=alpha2*x+beta2 l'equation de la droite passant par
n et perpendiculaire a la droite (pr)
       double alpha1=(q.x-p.x)/(double)(p.y-q.y);
       double beta1=mY-alpha1*mX;
       double alpha2=(r.x-p.x)/(double)(p.y-r.y);
       double beta2=nY-alpha2*nX;
         //le centre c du cercle est alors le point d'intersection
des deux droites ci-dessus
       cX=(beta2-beta1)/(double)(alpha1-alpha2);
       cY=alpha1*cX+beta1;
       cRadiusSquared=(p.x-cX)*(p.x-cX)+(p.y-cY)*(p.y-cY);
       if (cRadiusSquared>=resRadiusSquared) continue;
       boolean allHit = true;
```

```
        for (Point s: points)
          if ((s.x-cX)*(s.x-cX)+(s.y-cY)*(s.y-cY)>cRadiusSquared){
            allHit = false;
            break;
          }
                          if    (allHit)    {System.out.println("Found
r="+Math.sqrt(cRadiusSquared));resX=cX;resY=cY;resRadiusSquared=cR
adiusSquared;}
      }
    }
  }

                 return                  new                  Circle(new
Point((int)resX,(int)resY),(int)Math.sqrt(resRadiusSquared));
}
```