Étude sur l'algorithme Welzl résolvant le problème du cercle minimum

ZHAO Wenzhuo

Décembre 2020

Abstract

Étant donné un ensemble de points dans le plans, nous cherchons le cercle minimum unique qui contienne tous les points. Nous comparaisons la performance d'un algorithme naïf en temps quadratique à celle de l'algorithme de Welzl [4] qui calcule le cercle minimum d'une manière récursive en temps linéaire.

Mots-clés— cercle minimum, algorithme welzl, complexité

Contents

1 Introduction **2**
- 1.1 Définition . . . 2
  - 1.1.1 Unicité de cercle minimum . . . 2
  - 1.1.2 Notation . . . 2

2 Algorithmes **3**
- 2.1 Algorithme naïf . . . 3
  - 2.1.1 Pseudo Code . . . 4
- 2.2 Algorithme de Welzl . . . 4
  - 2.2.1 Pseudo Code . . . 4

3 Implémentation **5**
- 3.1 Structure de données . . . 5
- 3.2 Test . . . 6
- 3.3 Visualisation . . . 6

4 Performance **7**
- 4.1 En fonction de l'algorithme . . . 7
- 4.2 En fonction de la structure de données . . . 8

5 Notions en 3D **8**

6 Conclusion **8**
- 6.1 Amélioration ? . . . 9
- 6.2 Ce qu'il nous reste à faire . . . 9

1 Introduction

Dans plusieurs domaine d'industrie, il existe plusieurs problèmes qui se réduisent au calcul d'un cercle minimum couvrant un ensemble de points dans un plan, par exemple la détection de la collision d'objets dans les jeux vidéos ou la détermination d'un centre dans un réseaux de la télécommunication. Ce problème fût proposé par le mathématicien James Joseph Sylvester en 1857. Plusieurs solutions furent proposées pour calculer le cercle minimum, dans un temps linéaire, on peut citer que l'algorithme de Megiddo en 1983 qui est performant mais n'est pas évident. En 1991, Emo Welzl a proposé un algorithme probabiliste qui parcourt l'ensemble de points en agrandissant le cercle si un point aléatoire n'est pas dedans à chaque appel récursif. Nous étudions principalement cet algorithme et réaliser une étude sur la performance de l'algorithme sur différentes structure de données.

1.1 Définition

Un cercle est caractérisé par son point de centre et son rayon. Le cercle minimum pour un point est ce point lui-même. Pour un ensemble de plusieurs points, soit deux points qui constituent un segment où on peut trouver le centre et le rayon, soit passant par au moins trois points qui sont cocycliques.

1.1.1 Unicité de cercle minimum

Soit P l'ensemble de points, supposons qu'il existe deux cercles minimum contenant tout point de P, alors leur intersection contient aussi tout point de P. Supposons que ces deux cercles passent par vecteur $\vec{z_1}$ et $\vec{z_2}$, alors le cercle d'intersection est sur le centre $\frac{1}{2}(\vec{z_1} + \vec{z_2})$ qui a un rayon nécessairement plus petit. Nous avons une contradiction donc le cercle minimum contenant tout point d'un ensemble est unique.

1.1.2 Notation

Nous utilisons plusieurs fonctions mathématiques en complexité constante pour calculer un cercle. Pour distinguer le cas de deux points et de trois points, nous supposons que tous les points ne sont pas identiques.

1. Pour déterminer que trois points sont cocycliques, il nous faut vérifier que ces trois points ne sont pas colinéaires. Nous pouvons utiliser le produit vectoriel de deux vecteurs afin de vérifier cette contrainte. Soient P, Q, R trois points différents qui composent deux vecteurs $\vec{PQ}$, $\vec{PR}$, si $\vec{PQ} \times \vec{PR} \neq 0$, alors on peut dire que les trois points P, Q, R ne sont pas colinéaires.
   Cette fonction est nommée *crossProduct* dans la suite de ce rapport.

2. Si ces trois points sont cocycliques, nous pouvons calculer le cercle circonscrit du triangle composé par ces trois points. Soient $P(x_1, y_1)$, $Q(x_2, y_2)$, $R(x_3, y_3)$ trois points différents, soit le centre de cercle $O(x, y)$ qu'on va calculer, en fonction d'égalité de la distance entre ces trois points et le centre, nous pouvons déduire que

$$(x_1 - x)^2 + (y_1 - y)^2 = (x_2 - x)^2 + (y_2 - y)^2$$
$$(x_2 - x)^2 + (y_2 - y)^2 = (x_3 - x)^2 + (y_3 - y)^2$$

Après la simplification, nous avons:

$$2(x_2 - x_1)x + 2(y_2 - y_1)y = x_2^2 + y_2^2 - x_1^2 - y_1^2$$
$$2(x_3 - x_2)x + 2(y_3 - y_2)y = x_3^2 + y_3^2 - x_2^2 - y_2^2;$$

Soient

$$A_1 = 2(x_2 - x_1)$$
$$B_1 = 2(y_2 - y_1)$$
$$C_1 = x_2^2 + y_2^2 - x_1^2 - y_1^2$$
$$A_2 = 2(x_3 - x_2)$$
$$B_2 = 2(y_3 - y_2)$$
$$C_2 = x_3^2 + y_3^2 - x_2^2 - y_2^2;$$

alors on peut voir deux équations:

$$A_1x + B_1y = C_1;$$
$$A_2x + B_2y = C_2;$$

Selon la règle de Cramer, on peut calculer x et y:

$$x = (C_1B_2 - C_2B_1)/(A_1B_2 - A_2B_1)$$
$$y = (A_1C_2 - A_2C_1)/(A_1B_2 - A_2B_1)$$

Il suffit de calculer la distance entre P ou Q ou R et le centre O pour avoir le rayon.
Cette fonction est nommée $triangleCircumscribedCircle$ dans la suite de ce rapport.

3. Pour vérifier si un point est dans un cercle, il suffit de comparer la distance entre ce point et le centre du cercle.

2 Algorithmes

Nous allons introduisons un algorithme naïf et l'algorithme de Welzl.

2.1 Algorithme naïf

Supposons que cet algorithme est bien correct afin de nous servira à valider les résultats avec l'algorithme de Welzl à la suite.
Nous testons toutes les paires de point dans l'ensemble, essayant de trouver un cercle contienne tous les points. Si un tel cercle existe, alors il est minimal et nous pouvons retourner ce cercle.
L'opération de prendre chaque paire est en $O(n^2)$ et l'opération de tester l'appartenance de chaque point dans ce cercle est en $O(n)$, donc le cercle est trouvé en $O(n^3)$.
Si aucun cercle n'est trouvé dans la première étape, alors nous devons considérons le cercle passant au moins trois points de l'ensemble. L'opération de prendre un groupe de trois points est en $O(n^3)$ et l'opération de tester l'appartenance de chaque point dans ce cercle est en $O(n)$, donc le cercle est trouvé en $O(n^4)$. En pire cas, cet algorithme est en $O(n^3) + O(n^4) = O(n^4)$.

2.1.1 Pseudo Code

Algorithm 1: Cercle Minimum

```
Data: Collection<point> points
Result: Circle
Circle circle;
foreach Point p in points do
    foreach Point q in points do
        Point center ← middlePoint(p, q);
        Float radius ← distance(p, center);
        circle ← Circle(center, radius);
        if circle contains points then
            return circle;
foreach Point p in points do
    foreach Point q in points do
        foreach Point r in points do
            if crossProduct(p,q,r) = 0 then
                continue;
            Circle circleTemp ← triangleCircumscribedCircle(p, q, r);
            if circleTemp contains points then
                if circleTemp is smaller than circle or circle is null then
                    circle ← circleTemp;
return circle;
```

2.2 Algorithme de Welzl

Au départ, nous avons un ensemble de points P et un ensemble vide R. Nous enlevons à chaque appel récursif un point p de l'ensemble P jusqu'à :
- soit ne plus en avoir de points dans P. Si R contient 2 points, nous créons un cercle minimal passant par les deux points de R; sinon nous créons un cercle nul.
- soit R contient 3 points dans lequel cas nous créons un cercle circonscrit à ces trois points.

Lors de chaque remontée, nous testons si le cercle contient le point p. Si le cercle ne contient pas le point p, alors le point p est sur le contour du cercle et on recommence en ajoutant p à R.

2.2.1 Pseudo Code

Algorithm 2: Cercle Minimum

```
Data: Collection<point> points
Result: Circle
Collection<point> P = points;
Collection<point> R;
return welzl(P, R);
```

Algorithm 3: Welzl

```
Data: Collection<point> P, Collection<point> R
Result: Circle
Circle D;
if size(P) ≤ 1 or size(R) ≥ 3 then
    if isConcyclic(R) then
        D ← makeCircleWithPoints(R);
else
    choose a Point p randomly from P;
    remove p from P;
    D ← welzl(P, R);
    if D not contains p then
        add p to R;
        D ← welzl(P, R);
return D;
```

3 Implémentation

Nous utilisons le langage Java pour une programmation orienté objet. Il nous permet d'utiliser la bibliothèque JUnit pour tester l'identification des résultats de l'algorithme naïf et de l'algorithme de Welzl sur plusieurs instances. Une visualisation de cercle est aussi possible.

3.1 Structure de données

Selon notre implémentation de l'algorithme, nous utilisons les opérations l'ajout, la suppression et l'accès d'élément très souvent. Pour créer un cercle par la fonction *triangleCircumscribedCircle*, il est donc fréquent de prendre 3 éléments depuis l'ensemble. Le temps utilisé sur cette opération est important pour calculer le temps final.

ArrayList :
Une ArrayList en java est un tableau dynamique. Cette implémentation est une séquence d'objets autorisant les doublons, la valeur null et l'ajout à un index donné. Elle est basée sur un tableau dont la taille est gérée en interne. Les opérations l'ajout, la suppression est en $O(1)$ et l'accès d'un élément est réalisé par l'itération qui est en $O(n)$ avec la fonction Java `get(index)`.

Hashset :
Un HashSet en java est un ensemble d'objets non ordonnés, ne pouvant être triés. L'implémentation d'un HashSet est basée sur une HashMap. De suite, chaque objet sera insérer grâce à sa méthode HashCode. L'accès mémoire est constant. Les opérations l'ajout, la suppression est en $O(1)$ et l'accès d'un élément est réalisé par le HashCode qui est en $O(1)$. C'est donc plus rapide que ArrayList mais pour prendre des éléments, il nous faut créer un itérator qui utiliser la méthode `next()` prenant un élément. Nous sauvegardons cet élément dans un tableau de taille fixe. Une répétition de cette suite de ces opérations peut faire prendre plus de temps que l'on souhaite.

3.2 Test

Les fichiers source pour nos expérimentations proviens de *VAROUMAS* [1].
Avec le support de JUnit 4 et 5, le `@ParameterizedTest` est utilisé pour réaliser la séquence de 1664 test. Les fichiers sont lus et mis en place dans `Stream` pour servir à la méthode de test comme la source de données. Une telle méthode qui fournit le stream de données est marqué comme `@MethodSource`.Nous utilisons `assertEquals(circle, circle, delta)` dans chaque test unitaire.
Nous comparons les cercles retournés par les algorithmes par la position du point de centre et le rayon. En raison d'utilisation de division et l'incertitude de `random()`, une tolérance d'erreur moyenne est prévue. Pour 1664 tests à la taille de 256 points, 95% de tests passent avec une tolérance d'erreur de 4px sur la position de centre et la longueur de rayon.
En même temps, nous lançons un chronomètre en mille second. Le temps utilisé pour faire chaque test est écrit dans un fichier pour servir à dessiner le diagramme dans la suite de ce rapport.

3.3 Visualisation

En utilisant la bibliothèque `Java.awt`, une visualisation est possible qui nous permet de comparer de différents cercles retournés par nos implémentations d'algorithmes. Nous pouvons voir que les cercles sur les figures 1 et 2 sont quasiment identiques aux yeux.

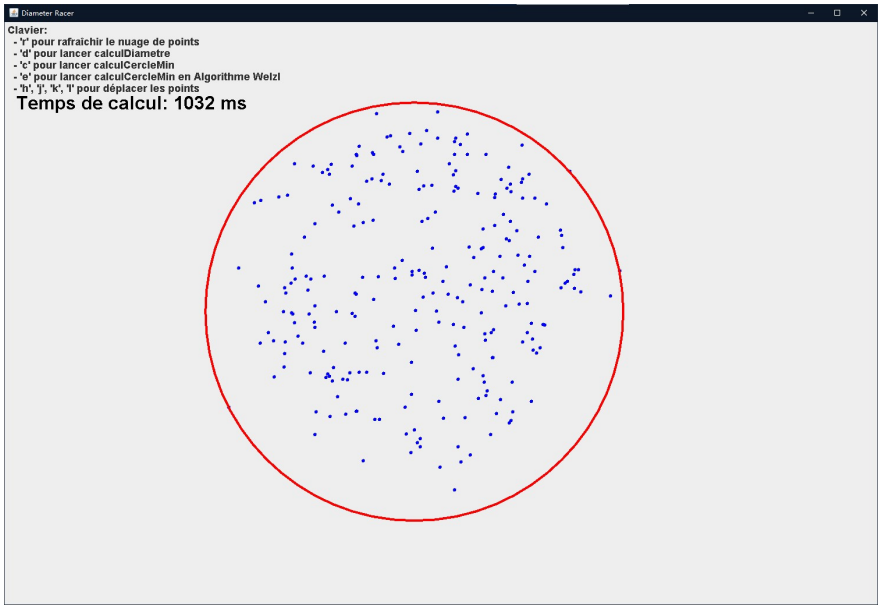

Figure 1: Cercle dessiné par l'algorithme naïf

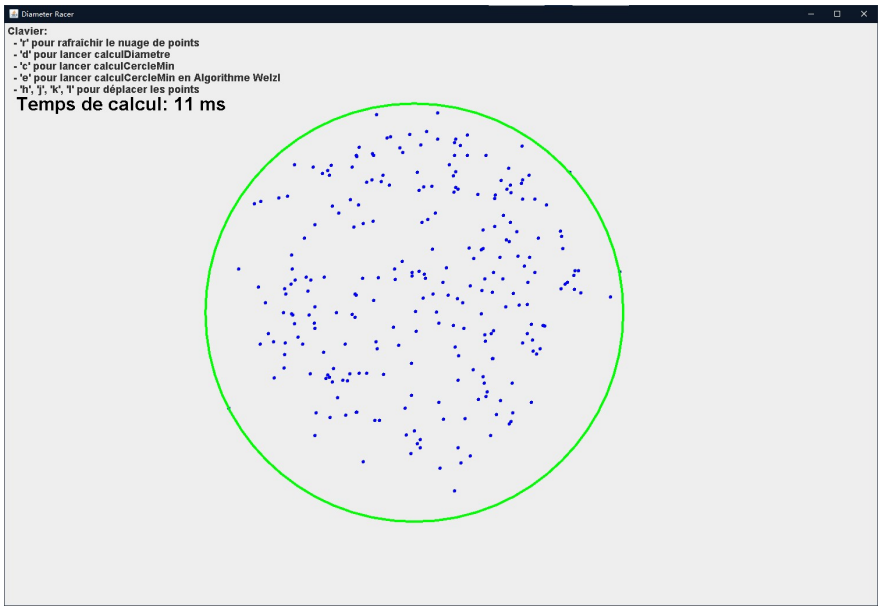

Figure 2: Cercle dessiné par l'algorithme de Welzl

4 Performance

4.1 En fonction de l'algorithme

Sur le figure 3, il est évident que l'algorithme naïf a un mauvais score par rapport à l'algorithme de Welzl. Pour le temps moyen de 1664 tests, l'algorithme naïf prend environ 991ms et l'algorithme de Welzl prends environs 0,35ms
Nos expérimentations admet que l'algorithme de Welzl est en temps linéaire comme prévu.

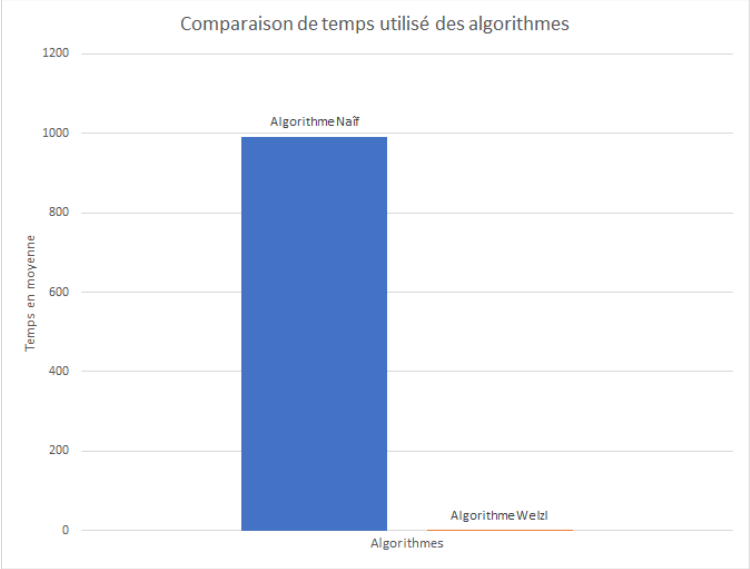

Figure 3: Temps en moyenne pour l'algorithme naïf et l'algorithme de Welzl

4.2 En fonction de la structure de données

Dû aux opérations pour l'accès en HashSet, et donc à un accès supérieur à $O(1)$, l'algorithme de Welzl réalisé avec HashSet en orange sur le figure 4 est moins performante qui prends un temps moyen environ 5ms sur 1664 tests, en plus d'avoir une variance importante. Celle de l'ArrayList est faible, avec une bonne performance.

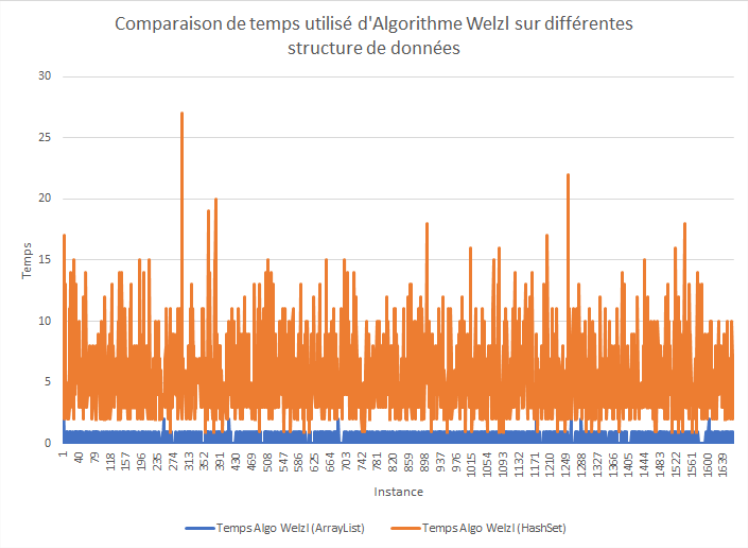

Figure 4: Temps en moyenne pour l'algorithme de Welzl avec ArrayList et HashSet

5 Notions en 3D

A la dimension de 3, nous pouvons démontrer que le sphère englobante minimum est unique, comme la démonstration 1.1.1. L'article [2] a décrit l'implémentation en C++ de cet algorithme à la dimension d.
Soit P, R deux sous ensemble de l'ensemble d'objets S. On a $P \cup R = S$ et $P \cap R = \emptyset$. Nous définissons que $MB(P, R)$ est le sphère minimum contenant tout point de P à l'intérieur et contenant tout point de R au contour. Similaire à l'algorithme de Welzl en 2D, nous avons $MB(S) = MB(S, \emptyset)$. Définissons $\overline{MB(R)} = MB(\emptyset, R)$, qui s'exprime que l'ensemble R est le sphère englobante minimum.
Le principe est similaire à l'algorithme de Welzl en 2D. Cet algorithme est très effective à la dimension d ($d \leq 30$).[3]

6 Conclusion

L'algorithme de Welzl est donc un algorithme performant, ou le plus performant algorithme à ce jour. Cet algorithme peut aider à résoudre de nombreux problèmes dans l'industrie. Il est possible de réaliser une meilleur implémentation en Java, voire sur des autres langages pour avoir une optimisation sur l'utilisation de mémoire. D'après nos expérimentations, l'implémentation de cet algorithme sur ArrayList est la meilleur choix.

6.1 Amélioration ?

Il est possible de réaliser une meilleur implémentation. Au début de l'appel de la fonction, il est obligatoire copier l'ensemble (autrement dit `clone()`) pour les modifications afin de ne pas modifier le même segment de mémoire. Pour un nombre important de points, une telle opération peut prendre une mémoire considérable donc survient une erreur StackOverFlow. Dans nos expérimentations, les instances de test sont tout à la taille de 256 points qui ne vont pas déclencher cette erreur donc heureusement on n'a pas à avoir beaucoup de connaissance sur Java ou architecture de système. Mais pour une application possible, il nous faut mieux organiser l'occupation de mémoire.

6.2 Ce qu'il nous reste à faire

Il nous semble difficile de faire mieux au niveau de la théorie, la seule amélioration possible est trouver le meilleur choix de point à retirer dans l'ensemble de départ au lieu d'une prise aléatoire de point. La méthode `move-to-front` dans l'article orignal peut améliorer la performance qui gagne le temps de générer une indice aléatoire, mais en raison du temps limité pendant cette étude, cette approche n'est pas demandée à réaliser donc nous ne l'avons pas explorée.

References

[1] *Base de tests*. `http://www-apr.lip6.fr/œbuixuan/files/algav2020/Varoumas_benchmark.zip`.

[2] Bernd Gärtner. *Fast and Robust Smallest Enclosing Balls*.

[3] Fischer Kaspar, Gärtner Bernd, and Kutz Martin. *Fast Smallest-Enclosing-Ball Computation in High Dimensions*.

[4] Emo Welzl. "Smallest enclosing disks (balls and ellipsoids)". In: *New Results and New Trends in Computer Science* ().