Algorithme Welzl résolvant le problème du cercle minimum

Guillaume Hivert

25 février 2016

Introduction

A travers ce rapport, le but sera d'étudier les différences entre deux algorithmes résolvant le même problème. Ce dernier se trouve être le cercle couvrant minimum d'un ensemble de points. Le premier algorithme utilisé est un algorithme naïf, se contentant de tester les points séquentiellement, tandis que le deuxième est l'algorithme de Welzl, un algorithme linéaire partant d'un cercle non optimal pour affiner son résultat et arriver à un resultat optimal.
Testé avec différentes instances de points, il est donc possible de comparer ces algorithmes, en vitesse ou en précision.

Les performances ont été testés sur Mac OS X 10.11, avec un Core i7.

Résultats

Théorie et complexité

Ainsi que précisé dans l'introduction, deux algorithmes ont été mis en place : l'algorithme de Welzl, et un algorithme naïf. C'est ce dernier qui sera traité en premier.

Chacun des deux algorithmes prends en entrée un ensemble de points dont il faut trouver un cercle couvrant minimum.

Algorithme naïf

L'algorithme naïf se compose de plusieurs boucles impliqués, parcourant l'ensemble des points de l'ensemble.

Dans un premier temps, on tente donc de trouver un cercle dont le centre est le milieu de deux points de l'ensemble, de rayon équivalent à la moitié de la distance entre les deux points. Si le cercle obtenu recouvre l'intégralité des points de l'ensemble, alors on stocke ce cercle comme un étant le candidat potentiel pour le cercle couvrant minimum de l'ensemble des points. Si l'on disposait déjà d'un cercle couvrant minimum potentiel, on vérifie que notre nouveau cercle est plus petit que celui-ci.
En parcourant l'ensemble des couples de points, on s'assure donc de trouver le plus petit cercle couvrant tous les points, s'il existe. Dans ce cas, on retourne donc le cercle trouvé.

Dans un second temps, on parcourt l'ensemble des groupes de trois points distincts deux à deux de l'ensemble, et on regarde si le cercle circonscrit au triangle composé des trois points couvre l'ensemble des points de l'ensemble. Si c'est le cas, on le stocke comme nouveau cercle couvrant minimum si l'ancien est plus grand que celui-ci. Sinon, on conserve l'ancien. Enfin, on retourne le cercle trouvé.

Il est indispensable d'effectuer le parcours des groupes de trois points de l'ensemble : en effet, on peut avoir un cas où deux points couvre la quasi-intégralité de l'ensemble, mais où quelques points ne sont pas inclus. Avec un groupe de trois points, il est possible de s'assurer que l'on considérera forcément le groupe de trois points les plus éloignés les uns des autres de l'ensemble. Dans ce cas, on trouvera forcément un cercle couvrant l'ensemble

des points de l'ensemble, bien qu'il puisse ne pas être minimum. Il faut donc parcourir l'ensemble des groupes de trois points distincts deux à deux.

L'algorithme en question peut s'écrire ainsi en C :

```
struct cl naive(struct pt points[], size_t size) {
  struct cl circle = circle_nil();

  // For two points.
  for (size_t i = 0; i < size; i++) {
    for (size_t j = 0; j < size; j++) {
      struct pt p1 = points[i], p2 = points[j];
      if (p1.equal(p1, p2)) continue;
      bool all_covered = true;
      struct cl temp = new_circle(middle(p1, p2), distance(p1, p2) / 2);
      for (size_t k = 0; k < size; k++)
        if (! temp.cover(temp, points[k])) {
          all_covered = false;
          break;
        }
      if (all_covered)
        if (temp.radius < circle.radius || circle.equal(circle, circle_nil()))
          circle = temp;
    }
  }
  if (! circle.equal(circle, circle_nil()))
    return circle;

  // For three points.
  for (size_t i = 0; i < size; i++) {
    for (size_t j = 0; j < size; j++) {
      struct pt p1 = points[i], p2 = points[j];
      if (p1.equal(p1, p2)) continue;
      for (size_t k = 0; k < size; k++) {
        struct pt p3 = points[k];
        if (p3.equal(p3, p1) || p3.equal(p3, p2)) continue;
        bool all_covered = true;
        struct cl temp = circum_circle(p1, p2, p3);
        for (size_t l = 0; l < size; l++)
          if (! temp.cover(temp, points[l])) {
            all_covered = false;
```

```
            break;
          }
        if (all_covered)
          if (temp.radius < circle.radius || circle.equal(circle, circle_nil()))
            circle = temp;
      }
    }
  }
  return circle;
}
```

On peut ainsi voir que la première partie de l'algorithme se compose de trois boucles for imbriquées les unes dans les autres, parcourant chacune l'ensemble des points de l'ensemble. Les instructions à l'intérieur des boucles sont toutes réalisées en $O(1)$. Si n est la taille de l'ensemble des points, alors la complexité de la première partie est en $O(n^3)$.
Pour la deuxième partie, on peut noter qu'il s'agit cette fois de quatre boucles for imbriquées les unes dans les autres, afin d'avoir un groupe de trois points de l'ensemble, puis le parcours de tous les points pour déterminer si le cercle est couvrant ou non. Chaque instruction contenu dans les boucles est en temps constant $O(1)$, y compris l'obtention du cercle circonscrit à un triangle. On a donc une complexité en $O(n^4)$ pour cette deuxième partie.

La complexité de l'algorithme naïf est donc en $O(n^3) + O(n^4)$, soit $O(n^4)$.

Algorithme de Welzl

L'algorithme de Welzl est un algorithme récursif prenant en entrée un ensemble de points dont il faut trouver le cercle couvrant, et un ensemble de points qu'il faut "rajouter" au cercle précedemment obtenu.
En effet, il commence par sélectionner un point au hasard de l'ensemble des points, qu'il' garde et qu'il' retire de l'ensemble. Il va ensuite effectuer un appel récursif sur l'ensemble de points ôté du point et le même ensemble de points à rajouter au cercle. Au début, cet ensemble est vide. Il va donc récursivment enlever tous les points de l'ensemble de départ, puis il va calculer le cercle couvrant de l'ensemble vide. Il obtient donc un cercle de rayon nul. Il part ensuite de ce cercle, et il va vérifier si le dernier point enlevé de l'ensemble est contenu dans le cercle. Puisque ce dernier a un rayon nul, il ne peut contenir le point. Il rajoute donc ce point à l'ensemble des points à ajouter, puis se rappelle récursivement. L'ensemble avec un point à rajouter

contient un point : il crée donc un cercle avec ce point comme centre, et un rayon nul, qu'il' renvoie.
Il arrive donc dans une configuration où il dispose d'un cercle avec un centre, mais un rayon nul : en testant si le point précedemment enlevé est contenu dans le cercle, il va donc ajouter ce point à l'ensemble des points à rajouter, et recommencer la récursion. Cette fois, l'ensemble de points à rajouter n'est plus vide, et l'algorithme va calculer un cercle entre ces deux points, qu'il renverra. En continuant ainsi, il obtient donc un cercle successivement affiné, en ajoutant des points dans la bordure des points à rajouter, qu'il va inclure petit à petit en élargissant le cercle. On obtient donc un cercle minimum couvrant pour l'ensemble des points.

l'algorithme peut s'écrire ainsi en C :

```
struct cl _welzl(struct pt points[], struct pt border[],
                 size_t size_points, size_t size_border) {
  // Duplicates points.
  struct pt *pts = calloc(512, sizeof *pts);
  struct pt *bor = calloc(512, sizeof *bor);

  memcpy(pts, points, sizeof *pts * size_points);
  memcpy(bor, border, sizeof *bor * size_border);

  points = pts;
  border = bor;

  // Initiliazes the empty circle.
  struct cl circle;

  // And begins.
  if (size_points == 0 || size_border == 3) {
    circle = b_md(border, size_border);
  } else {
    // Pick a random point and remove it from points.
    size_t alea    = nxt_i(size_points);
    struct pt back = points[alea];
    points[alea]   = points[--size_points];

    // Recursive call.
    circle = _welzl(points, border, size_points, size_border);
```

```
    // Test if another recursive call is needed.
    if (circle.equal(circle, new_circle(new_point(0, 0), -2)) ||
        !circle.contains(circle, back)) {
      border[size_border++] = back;
      circle = _welzl(points, border, size_points, size_border);
    }
  }

  // And free the memory and return.
  free(points);
  free(border);
  return circle;
}

struct cl welzl(struct pt points[], size_t size) {
  struct pt *start = malloc(sizeof *start * size);
  return _welzl(points, start, size, 0);
}
```

La complexité de l'algorithme est cette fois plus raisonnable : pour l'ensemble de départ, un élément est enlevé, puis deux appels récursifs sont effectués dans le pire des cas. Chacun de ces appels récursifs peut effectuer au maximum deux appels récursifs. Pour chaque élément de l'ensemble de départ, il y a donc au maximum 2 itérations. Enfin, chaque instruction dans les appels de fonctions est en temps constant, soit $O(1)$. Toutefois, il est possible de montrer que pour le point i, celui-ci a une probabilité $O(1/i)$ de provoquer un appel récursif. Il tend donc à n'y avoir qu'un seul appel récursif.

La complexité est donc en $O(n)$.

Expérimentations

Chacun des algorithmes a été testé sur la base de test Varoumas. Cela représente des ensembles de points variés, dont il faut trouver le cercle couvrant minimum. Bien que les algorithmes retournent le cercle couvrant minimum optimal, leur complexité, en terme d'espace comme de temps n'est pas la même. En effet, des différents tests, il ressort que l'algorithme de Welzl se trouve être bien plus efficace en temps :

Chaque croix verte représente une instance, et indique le temps (en secondes)

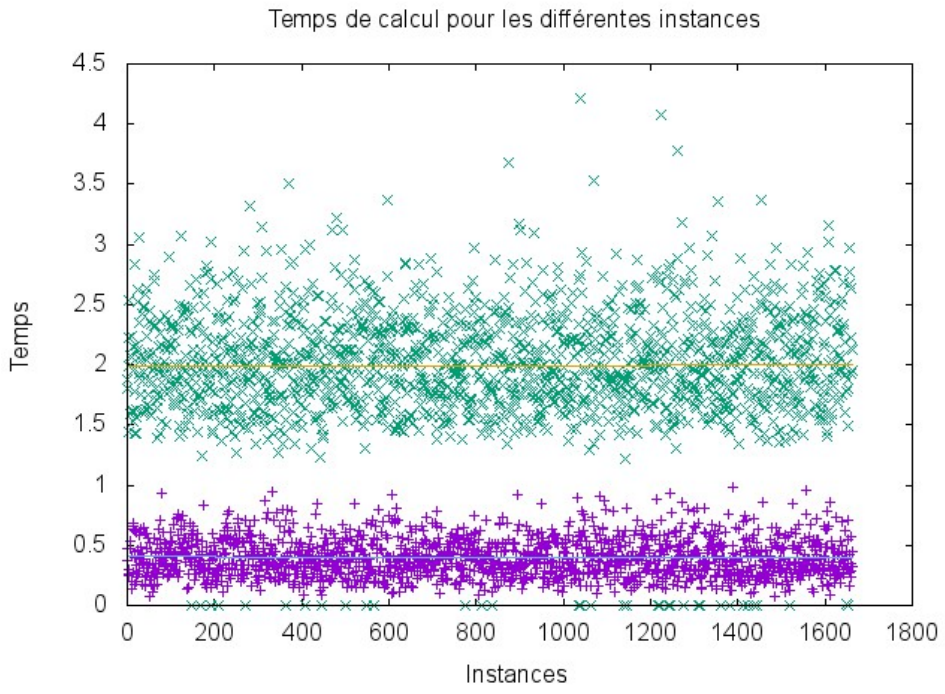

Figure 1: Temps de calcul pour les différentes instances

pris pour le calcul du cercle. On peut voir en jaune la moyenne linéaire du temps pris par l'algorithme naïf pour calculer une instance. Certains éléments sont légèrement en dehors, cela peut être dû au fait que l'on ne maitrise pas complètement l'environnement, et un calcul lourd en parallèle peut affecter les résultats de celui-ci. Il est donc intéressant de noter le résultat moyen.

Chaque croix violette représente le temps de calcul pris par une instance avec l'algorithme de Welzl. Il est possible cette fois de voir une forte diminution du temps de traitement pour chaque instance. On passe en moyenne (la ligne bleue) de 2 secondes à 0,5 secondes. Soit une division par 4 du temps de calcul pour chaque instance. Encore une fois, un calcul lourd en parallèle peut affecter les performances de l'algorithme, et expliquer les différences entre certaines instances et d'autres. Il faut également noter que l'algorithme de Welzl repose en partie sur de l'aléatoire. Puisque cela change à chaque itération, il est possible qu'une même instance ne prenne pas le même temps de calcul de fois de suite, puisqu'à chaque appel , les points sont tirés différemment, pouvant avantager ou désavantager certaines instances.

Discussions

Comme il a été vu, chaque algorithme retourne le cercle minimum couvrant optimal, et s'effectue ici dans un temps relativement réduit. En effet, l'ensemble des instances de Varoumas a pu être calculé en 40 minutes environ, puisque l'ensemble des calculs s'est fait en C. Le C étant un langage de performance par nature, l'algorithme s'exécute très rapidement, et la différence entre l'algorithme naïf et l'algorithme de Welzl est finalement assez peu important : bien que le rapport soit de 4, le temps d'exécution de l'algorithme naïf, en général de 2 secondes, est très abordable.
Les mêmes algorithmes ont été écrits en Ruby également, pour sa plus grande aisance pour la manipulation des ensembles. Les résultats sont alors assez différents : alors que l'algorithme de Welzl ne prenne pas beaucoup plus de temps (là encore en moyenne 0,5 secondes), l'agorithme naïf prends pratiquement 100 secondes à chaque instance pour résoudre le problème. Alors qu'en C, la différence est minime (un facteur 4), en Ruby, la différence est un facteur de 200. 40 minutes ne suffisent plus pour résoudre la base de test avec l'algorithme naïf : il faut un peu plus d'une journée.

On observe donc effectivement la différence de complexités entre ces deux algorithmes : le passage de $O(n^4)$ à $O(n)$ réduit, pour le C, d'un facteur 4, et pour le Ruby d'un facteur 200. Cela est plus marqué pour le Ruby, puisque le langage tourne sur une machine virtuelle. Chaque instruction Ruby demande donc bien plus d'instructions réelles au processeur, multipliant encore le temps de calcul.

Bien que ce gain soit non négligeable, et fortement en faveur de l'algorithme de Welzl, il peut être encore accentué en augmentant la taille des instances de points : la taille des instances de la base de test Varoumas se compose d'ensembles de points de taille 256. Ainsi, on obtient environ 256 appels récursifs en utilisant Welzl, et environ 256^4 (4 294 967 296) opérations en utilisant l'algorithme naïf. En augmentant la taille des ensembles, l'écart se creuse d'autant plus : avec 512 points, là où Welzl effectue environ 512 appels récursifs, l'algorithme naïf effectue 512^4 (68 719 476 736) opérations environ. Le temps pris pour effectuer un calcul de cette taille devient alors très lourd pour un langage comme le C, et inabordable pour un langage comme le Ruby. L'algorithme naïf devient donc rapidement inutilisable pour des instances suffisament grosses.

Il faut toutefois noter que l'algorithme de Welzl est récursif. Il faut donc, à chaque appel récursif, dupliquer l'intégralité des structures en mémoire.

Sur des instances de taille restreintes, cela ne pose pas de problèmes : la consommation mémoire n'explose pas. En revanche, avec des instances de taille supérieures, on peut rapidement se retrouver à consommer de nombreuses ressources, afin de conserver les ensembles des appels récursifs. Sans compter que chaque appel récursif prends de la place en pile afin de garder une trace de la fonction appelante, celle-ci n'étant pas récursive terminale.
Avec des ressources limitées, il est possible de se retrouver dans une sirtuation où il peut être compliqué d'exécuter l'algorithme, voire impossible si les instances sont trop importantes.

Il faut toutefois pondérer ces problèmes, dans la mesure où ces algorithmes seront rarement appelés dans un cadre avec peu de ressources disponibles : il est donc possible de se permettre de consommer de la mémoire afin de gagner en mémoire, à fortiori dans un domaine comme un jeu vidéo : il est plus important d'obtenir un résultat rapidement, même si celui-ci peut être légèrement faussé, plutôt qu'obtenir un résultat lent au dépend de la mémoire.

Conclusions

Pour conclure, il sera donc globalement plus intéressant d'utiliser l'algorithme de Welzl en règle général. Il est toutefois intéressant de noter que celui-ci peut également être pris en défaut. Il n'existe donc pas d'algorithme parfait, et il faut savoir trouver le meilleur compromis entre les deux. En effet, l'algorithme naïf – bien que peu consommateur de ressources – est beaucoup trop lent pour tourner sur de grosses instances, à fortiori avec un langage de programmation de haut niveau. L'algorithme de Welzl est lui bien plus efficace, mais peut se révéler gourmand en mémoire, et faire exploser la consommation mémoire sur des machines avec des ressources limitées.

Annexes

Compiler et exécuter le programme

Afin de compiler le programme en C, il faut se rendre dans le dossier Project/src, et effectuer un ./make (après avoir donné les droits d'exécution si nécessaire). Il compilera deux exécutables naive et welzl. Chaque programme tourne avec les mêmes options :

```
./exec folder [format file]
```

exec étant l'exécutable, folder, le dossier contenant les fichiers de test au format file.points (ici, le dossier samples), format étant pretty ou bench, et file le nom du fichier de sortie.
Un format en pretty sortira un fichier en pretty printing, là où un format en bench sortira un fichier csv.
Les arguments format et file sont optionnels, et par défaut sur pretty et "results_[nom de l'exec].csv".
Par exemple :

```
./make
./naive ../samples
```