Descrição, Implementação e Análise de Complexidade do Algoritmo QuickHull

Paulo Barbosa

Janeiro de 2021

Resumo

Os invólucros convexos são uma das estruturas mais ubíquas em geometria computacional. Ao longo dos anos, vários algoritmos para a computação destes foram surgindo. Aqui é apresentado um desses algoritmos: o QuickHull. Baseado no algoritmo de ordenação QuickSort, o QuickHull é, do mesmo modo, recursivo.
Na secção 1 é feita uma breve introdução aos invólucros convexos, à sua relevância e algumas aplicações.
Na secção 2 serão apresentadas algumas definições necessárias para entender como surge e funciona o algoritmo em questão.
Na secção 3 é apresentada a estratégia do algoritmo, bem como o seu pseudo-código.
Por último, na secção 4 é feita a sua análise de complexidade.

1 Introdução

Os invólucros convexos são uma das estruturas mais ubíquas em geometria computacional. Um dos primeiros artigos identificados como sendo desta área consistia na computação do invólucro convexo. Desde então, a área tem crescido e vários algoritmos para a computação deste foram surgindo.

Aqui é apresentado um desses algoritmos: o QuickHull.

Além disso, será feita a sua implementação na linguagem de programação C, bem como a análise de complexidade (melhor e pior caso).

De seguida, é apresentada uma imagem onde é possível observar o efeito da aplicação de um algoritmo análogo ao aqui apresentado sobre um conjunto de pontos em duas dimensões.

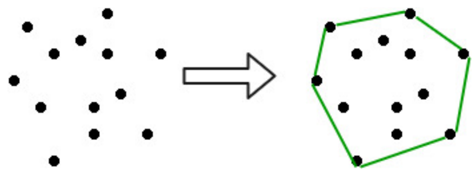

Figura 1: Aplicação de um invólucro convexo sobre um conjunto de pontos no plano R^2 (foto de GeeksForGeeks).

Antes de continuarmos com a geometria, serão apresentadas algumas aplicações (como descritas em [1]).

1. **Prevenção de colisão**: Se o invólucro convexo de um robô não colidir com nenhum obstáculo, este último também não o fará. Como a computação de caminhos que previnam a colisão é mais simples com um robô convexo, frequentemente recorre-se a um algoritmo de invólucro convexo.
2. **Análise de formas**: Formas poderão ser classificadas de acordo com as suas CDTs (Convex Deficiency Trees), estruturas que dependem para a sua computação num algoritmo de invólucro convexo.

2 Definições sobre Convexividade

Definição 1. *Sejam $v_0, ..., v_{n-1}$ n pontos distintos do plano R^2, com $n \geq 3$. Por convenção, considere-se que $v_i = v_j$ se $i \equiv j \mod n$.*

Seja e_i o segmento de reta que une v_i e v_{i+1}. Este é dado pela seguinte condição:

$$e_i = v_i v_{i+1} = \{v_i + t(v_{i+1} - v_i) \mid t \in [0, 1]\}$$

Analogamente ao que acontece com os pontos, os segmentos também são congruentes módulo n, ou seja, $e_0 = e_0$, $e_1 = e_{n+1}$, etc.

Definição 2. *Seja $A \subseteq R^n$ um conjunto. A diz-se **convexo** se:*

$$\forall_{a,b \in A} : ab \subseteq A$$

isto é, se $\forall_{t \in [0,1]} : a + t(b - a) \in A$.

Definição 3. *Sejam $x_1, ..., x_k \in R^n$. Uma **combinação convexa** de $x_1, ..., x_k$ é um elemento de R^n da forma*

$$t_1 x_1 + ... + t_k x_k$$

em que $\forall_i : t_i \geq 0$ e $\sum_{i=1}^{k} t_i = 1$.

Definição 4. *O **invólucro convexo** de um conjunto $S \subseteq R^n$ é o conjunto de todas as combinações convexas de elementos de S.*

Proposição 1. *O invólucro convexo de $S \subseteq R^n$ é o menor conjunto convexo que contem S.*

3 O Algoritmo QuickHull

O algoritmo QuickHull permite computar o invólucro convexo de um conjunto de pontos (passados como *input*) num espaço N dimensional. Será aqui apresentado o código no caso de $N = 2$.

O seu nome deve-se ao facto de ser baseado no algoritmo QuickSort (explicação em [2]). É, também, recursivo.

Antes de ser apresentado o pseudo-código, serão definidos os conceitos de pontos e arestas extremas, bem como apresentada a ideia geral do funcionamento do algoritmo.

Definição 5. *Os **pontos extremos** de um conjunto S de pontos no plano são os vértices de um invólucro convexo nos quais o ângulo interior é estritamente convexo, menor do que π.*

Definição 6. *Diz-se que uma aresta é uma **aresta extrema** se todos os pontos de um conjunto S estiverem sobre ou a um lado da linha determinada pela aresta. Considere-se o lado esquerdo como sendo o lado interior. Então, uma aresta não é extrema se houver algum ponto que não esteja à esquerda de ou nesta.*

A ideia principal do algoritmo é de usar uma aresta extrema como âncora para encontrar a próxima.

Primeiramente, são selecionados dois pontos extremos, x e y que correspondem, respetivamente, ao ponto mais baixo à direita e ao ponto mais alto à esquerda.

O ínvolucro convexo é, então, composto por um "invólucro superior" acima do segmento xy e por um "invólucro inferior" abaixo de xy.

Para um terceiro ponto extremo - z - estritamente à direita de xy, podemos descartar todos os pontos interiores ao triângulo $\Delta(x, y, z)$. De modo a optimizar o tempo de execução do algoritmo, o ponto z escolhido é aquele que maximiza a distância ao segmento xy.

Operando recursivamente é possível obter o invólucro convexo do conjunto de pontos inicial.

É, agora, apresentado o pseudo-código do algoritmo QuickHull. O código escrito em C encontra-se online em https://github.com/PJBarbosa98/QuickHull.

Considere-se S um conjunto de pontos em R^2 e $a, b \in S$.

Algorithm 1 QuickHull (a, b, S)

if S is empty **then**
    Devolver **NULL**.
end if
$c \leftarrow$ índice do ponto cuja distância ao segmento ab é máxima.
$A \leftarrow$ pontos estritamente à dirieta de ac.
$B \leftarrow$ pontos estritamente à direita de cb.
Devolver **QuickHull** $(a,\ c,\ A)$ + (c) + **QuickHull** $(c,\ b,\ B)$.

Algorithm 2 Main

$S \leftarrow$ conjunto de pontos 2D.
$x \leftarrow$ ponto mais baixo à direita de S.
$y \leftarrow$ ponto mais alto à esquerda de S.
$S_1 \leftarrow$ conjunto de pontos estritamente acima de xy.
$S_2 \leftarrow$ conjunto de pontos estritamente abaixo de xy.
Devolver (x) + **QuickHull** $(x,\ y,\ S_1)$ + **QuickHull** $(y,\ x,\ S_2)$

4 Análise de Complexidade

Por último, é apresentada a análise de complexidade para o algoritmo QuickHull.

Considere-se S um conjunto de pontos em R^2 tal que $|S| = n$, para um dado $n \in R$. Então,

- Encontrar os extremos x, y e partir S em S_1 e S_2 corre em complexidade $O(n)$.
- São necessários n passos para determinar o ponto extremo c, o que resulta numa complexidade $O(n)$.

Até agora, verifica-se o tempo de execução $O(n) + O(n) = O(n)$.

Resta, apenas, verificar a complexidade no passo recursivo.
Sabe-se que, na chamada do algoritmo, obtemos dois conjuntos A e B. Tome-se $|A| = \alpha$ e $|B| = \beta$, com $\alpha + \beta \leq n - 1$. Então, a complexidade total pode ser exprimida através de uma função T dada por $T(n) = O(n) + T(\alpha) + T(\beta)$.

No **melhor caso**, temos que A e B estão balanceados, ou seja, $\alpha = \beta = \frac{n}{2}$. Como resultado, $T(n) = O(n) + 2T(n/2)$. Logo, neste caso, a complexidade total é dada por $O(n \cdot log(n))$.

No **pior caso**, A e B estão o mais desbalanceados possível, isto é, $\alpha = 0$ e $\beta = n - 1$, ou vice-versa. Então, $T(n) = O(n) + T(n-1) = c \cdot n + T(n-1)$.

A expansão desta formula resulta em $O\left(n^2\right)$. Logo, neste caso, a complexidade total é dada por $O\left(n^2\right)$.

Referências

[1] Joseph O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1998.

[2] Robert Sedgewick. *Algorithms, 4th Edition.* Addison-Wesley Professional, 2011.