Algoritmo para gerar invólucros convexos

Artur Queiroz - PG38014
Luís Albuquerque - PG38015

17 de Janeiro de 2020

1 Introdução

Neste trabalho abordamos um dos principais temas de Geometria Computacional, Invólucros Convexos. Estes são muito usados porque têm aplicações em bastantes áreas como reconhecimento de padrões, processamento de imagem, estatística, etc.

Existem várias formas de construir um, mas neste trabalho vamos nos cingir a implementar o "*merge-hull*"que é um algoritmo que se usa, umas das técnicas mais importantes na computação, que se chama "Dividir para conquistar". Baseia-se em dividir um problema complexo, em problemas mais pequenos e mais faceis.

1.1 Notações utilizadas

$< \forall x : p(x) : q(x) >$ é equivalente a $\forall_{p(x)}.q(x)$

Quando fazemos
$\mathcal{R}\{$Descrição$\}$
significa que a "Descrição"é a razão pela qual a relação $\mathcal{R}$ é verdade.

left(l, p) : "o ponto p está à esquerda da linha l"
leftOn(l, p) : "o ponto p está à esquerda ou na linha l"
right(l, p) : "o ponto p está à direita da linha l"
rightOn(l, p) : "o ponto p está à direita ou na linha l"

2 Descrição

Para implementarmos o algoritmo apenas temos que seguir os passos a baixo.

- Ordenar os pontos por ordem lexicográfica.
- Remover os pontos repetidos.
- Criar o Invólucro convexo
  - Se o conjunto tem apenas 1 elemento, criar um "polígono"de 1 elemento e acaba-se a função.
  - Separar os pontos em dois conjuntos A e B, onde A contém os pontos da esquerda e B os da direita.
  - Calcular o invólucro convexo de A, $\mathcal{A} = \mathrm{I(A)}$ e o de B, $\mathcal{B} = \mathrm{I(B)}$ recursivamente
  - Encontrar os pontos de junção (tangentes).
  - Juntar $\mathcal{A}$ e $\mathcal{B}$, calculando o invólucro convexo de $\mathrm{A} \cup \mathrm{B}$.

Agora vamos explicar com mais detalhe todo o processo.

2.1 Ordenar os Pontos

Para ordenar os pontos, pode ser usado qualquer algoritmo de ordenação, tendo em atenção que a escolha do algoritmo de ordenação, pode alterar a complexidade do algoritmo como um todo. Nós optamos por escolher o algoritmo de ordenação *merge sort*, além de ter uma das melhores complexidades $\Theta(n \log n)$, achamos que se enquadra perfeitamente no espirito do algoritmo, "Dividir para conquistar".

2.1.1 Descrição de MergeSort

Input: array, índice esquerdo, índice direito

Começando com o índice esquerdo a 0, e o índice direito a (*tamanho do array*) - 1

- Primeiro encontra-se o índice médio do Array e divide-se em dois. (meio = (esquerda + direita)/2)
- Calcular o MergeSort(array, esquerda, meio) (a lista que fica à esquerda)
- Calcular o MergeSort(array, meio+1, direita) (a lista que fica à direita)
- No final junta os dois de forma ordenada.

2.2 Remover os Pontos Repetidos

Já que neste contexto sabemos que o array está ordenado, podemos fazer este algoritmo com 2 índices, um para percorrer o array, o outro para percorrer o novo array onde todos os elementos são diferentes. Se o valor do array nos dois índices for diferente então incrementa-se o segundo índice e atribui-se o valor do primeiro índice no valor do segundo índice. Se o valor dos dois índices for igual, apenas incrementa-se o primeiro índice.

2.3 Separar os Pontos em dois conjuntos

No trabalho decidimos usar um array, para que a sua divisão a meio tivesse uma complexidade constante é tão simples como encontrar o índice a meio do array.

2.4 Encontrar as tangentes

Na função que trata de encontrar os pontos que descrevem a tangente superior, começa-se com a linha do maior vértice de $I(S_1)$ ao menor vértice de $I(S_2)$ (lexicograficamente). A partir do vértice atual de $I(S_1)$ segue-se para o anterior, o mais longe possível, até que o vértice antecessor esteja estritamente à esquerda da linha, de seguida fazemos o dual para o vértice de $I(S_2)$. Repetindo este processo até não se conseguir mais ou encontrarmos um dos pontos extremos (o ponto mais pequeno ou o ponto maior, lexicograficamente).

Para encontrar a tangente inferior é o dual de encontrar a superior.

2.5 Juntar os invólucros convexos

Liberta-se a memória dos vértices entre as duas ligações e de seguida conecta-se as duas ligações.

3 Correção

Depois de mostrarmos como é o algoritmo, aqui vamos provar, porque é que o algoritmo faz o que diz que faz. Tendo I como a nossa função descrita em cima.

Queremos que no fim de execução do algoritmo, o poligono retornado seja

o invólucro convexo do conjunto dado no input.
Para isso vamos usar um Corolário dado nas aulas, descrito como:

"Seja $S \subseteq \mathbb{R}^2$ e P um poligono de vértices positivamente orientados $v_0, ..., v_{n-1} \in S$ tal que cada vértice é estritamente convexo e $S \subseteq P$. Então P é o invólucro convexo de S."

E a definição:

"v_i diz-se estritamente convexo se $\mathcal{A}(v_{i-1}, v_i, v_{i+1}) > 0$."

Seja o input S, conjunto de pontos, e o output P, poligono.
Caso Base ($S = \emptyset$):
Trivial

Caso Base ($\#S = 1$):
Seja $v_0 \in S$, então pela definição de I, $v_0 \in P$

Caso Indutivo:
Seja $S = S_1 \cup S_2$ tal que todos os pontos de S_1 sejam menores que os pontos de S_2 (lexicograficamente); $P_1 = I(S_1)$ e $P_2 = I(S_2)$ em que P_1 e P_2 são invólucros convexos por indução.

Suponhamos que os pontos de S são colineares.
Pelas definições de 2.4 vão encontrar as linhas $\overline{pq}$ e $\overline{qp}$, respetivamente, tal que p é o ponto menor de S, e q o ponto maior de S. Logo $P = \overline{pq}$, o invólucro convexo de S.

Suponhamos que os pontos de S não são colineares.
Agora para provar que P é o invólucro convexo do conjunto S, basta provar que P, com os vértices positivamente orientados $v_0, ..., v_{n-1}$, $\mathbf{v_0}, ..., \mathbf{v_{n-1}} \in \mathbf{S}$, **todos os vértices são estritamente convexos** e **todos os vértices estão no poligono**.

3.1 $\mathbf{v_0}, ..., \mathbf{v_{n-1}} \in \mathbf{S}$

Hipoteses Indutivas:

1. $pontos(P_1) \subseteq S_1$

2. $pontos(P_2) \subseteq S_2$

Como ao juntar os dois invólucros convexos P_1 e P_2 nunca acrescentamos pontos a P, além de pontos de P_1 e P_2, significa que os pontos de P estão contidos na união dos pontos de P_1 e pontos de P_2. Logo temos que $pontos(P) \subseteq pontos(P_1) \cup pontos(P_2)$.

$pontos(P)$
$\subseteq$ {Provado acima}
$pontos(P_1) \cup pontos(P_2)$
$\subseteq$ {Hipoteses de indução}
$S_1 \cup S_2$
$=$ {Definição de S}
S

3.2 Todos os vértices são estritamente convexos

Hipoteses Indutivas:

1. $< \forall p_i : p_i \in I(S_1) : \mathcal{A}(p_{i-1}, p_i, p_{i+1}) > 0 >$
2. $< \forall p_i : p_i \in I(S_2) : \mathcal{A}(p_{i-1}, p_i, p_{i+1}) > 0 >$

Pela descrição da "2.4 Encontrar as tangentes"

Quando não conseguirmos mais, significa que encontramos 2 vértices p_i e q_i, tais que:

$left(\overline{p_i q_i}, p_{i-1}) \wedge left(\overline{p_i q_i}, q_{i+1})$
$\equiv$ { definição de left }
$\mathcal{A}(p_i, q_i, p_{i-1}) > 0 \wedge \mathcal{A}(p_i, q_i, q_{i+1}) > 0$
$\equiv$ { regra dada na aula }
$\mathcal{A}(p_{i-1}, p_i, q_i) > 0 \wedge \mathcal{A}(p_i, q_i, q_{i+1}) > 0$

e arranjamos 2 vértices p_i e q_i, tais que:

$left(\overline{q_i p_i}, p_{i+1}) \wedge left(\overline{q_i p_i}, q_{i-1})$
$\equiv$ { definição de left }
$\mathcal{A}(q_i, p_i, p_{i+1}) > 0 \wedge \mathcal{A}(q_i, p_i, q_{i-1}) > 0$
$\equiv$ { regra dada na aula }
$\mathcal{A}(q_i, p_i, p_{i+1}) > 0 \wedge \mathcal{A}(q_{i-1}, q_i, p_i) > 0$

Com isto temos que os vértices encontrados que foram usados para juntar os dois invólucros são estritamente convexos no $I(S)$. Como por indução sabemos que os outros vértices são estritamente convexos, significa que todos os vértices em $I(S)$ são estritamente convexos.

3.3 Todos os vértices estão no poligono

Hipoteses Indutivas:

1. $S_1 \subseteq P_1$
2. $S_2 \subseteq P_2$.

Seja p_i o menor ponto da interseção da tangente superior com P_1, e q_i o maior ponto da interseção com P_2. Analogamente p_j o menor ponto da interseção da tangente inferior com P_1 e q_j o maior ponto da interseção com P_2.

Depois de se juntar P_1 e P_2 cria-se um invólucro convexo, onde se pode dividir em 3 partes.

- A: Todos os pontos de S_1 à esquerda de $p_j p_i$
- B: Poligono formado pelos pontos $\{p_i, p_j, q_j, q_i\}$
- C: Todos os pontos de S_2 à direita de $q_j q_i$

Ou seja $P = A \cup B \cup C$.
Se $S_1 \subseteq A \cup B$ e $S_2 \subseteq B \cup C$, então podemos concluir que $S_1 \cup S_2 = S \subseteq P$.
Por isso queremos provar que:

1. $S_1 \subseteq A \cup B$
2. $S_2 \subseteq B \cup C$

Seja A' são os pontos de S_1 à direita de $p_i p_j$, temos que $S_1 = A \cup A'$.
Queremos provar que $A' \subseteq B$.
Por definição B contém os pontos à direita de $p_j p_i$ e à esquerda de $q_j q_i$ e como os pontos são ordenados e divididos em P_1 e P_2 temos a garantia que a linha $q_j q_i$ está à direita de $p_j p_i$. Uma vez que $q_i p_i$ é uma tangente superior e $p_j q_j$ é uma tangente inferior, significa que todos os pontos de A' estão à esquerda de ambas essas tangentes. Logo $A' \subseteq B$. Concluindo que $S_1 = A \cup A' \subseteq A \cup B$

Analogamete para $S_2 \subseteq B \cup C$ ■

3.4 Proposição Tangente

Seja I um invólucro convexo e l uma linha que interseta num vértice de I, p_i, no sentido contrário dos ponteiros do relógio. Seja p_{i-1} o vértice anterior a p_i e p_{i+1} o vértice posterior a p_i. Temos que:

$$leftOn(l, p_{i-1}) \wedge leftOn(l, p_{i+1}) \Rightarrow < \forall p : p \in I : leftOn(l, p) >$$

3.4.1 Demonstração

Suponhamos que $leftOn(l, p_{i-1}) \wedge leftOn(l, p_{i+1})$ e que existe um $p \in I$ tal que $right(l, p)$ isso significa que a linha $\overline{p\ p_i}$ não está no invólucro convexo. Que é uma contradição, ou seja, $< \forall p : p \in I : leftOn(l, p) >$

■

4 Complexidade

A nossa implementação não foi exatamente igual ao algoritmo original, apesar de não alterar na conta da complexidade assintoticamente. Por isso vamos avaliar a correção da nossa implementação, e quando achamos pertinente, vamos fazer a ressalva, mensionando as diferenças em relação ao algoritmo original.

- Ordenar os pontos pela cordenada x, tem Complexidade $\Theta(n \log n)$
- Separar os pontos em dois conjuntos A e B, onde A contém os pontos da esquerda e B os da direita.
- Calcular o invólucro convexo de A, $\mathcal{A}$ = I(A) e o de B, $\mathcal{B}$ = I(B) recursivamente
- No final juntar $\mathcal{A}$ e $\mathcal{B}$, calculando o invólucro convexo de A $\cup$ B.

4.1 Ordenar pontos

Pela complexidade já conhecida do *merge sort*, temos que a complexidade de Ordenar pontos é:

$$\Theta(n \log n)$$

4.2 Remover pontos repetidos

Como os pontos são recebidos ordenados lexicograficamente, significa que podemos eliminar os pontos iguais com apenas uma simples passagem pelo array, tendo em conta apenas o índice do novo array e o indice do array antigo, logo a complexidade é:

$$\mathcal{O}(n)$$

4.3 Separar conjuntos de pontos

Para isso basta encontrar o meio do array, e como temos o tamanho do array é simples, faz-se apenas com uma operação, divisão por 2, o que torna a complexidade desta operação:

$$\mathcal{O}(1)$$

4.4 Encontrar tanjentes

Para encontrar a tangente inferior(superior), o pior caso é quando encontra os pontos limite (o ponto mais à esquerda e mais à direita do conjunto de pontos) simultaneamente na tangente inferior(superior), para isso tem que percorrer todos os pontos, o que torna a sua complexidade:

$$\mathcal{O}(n)$$

4.5 Juntar os invólucros convexos

Como para os poligonos usamos uma lista duplamente ligada e para juntar recebemos os 4 pontos a juntar, é só preciso "ligar"os pontos com 8 apontadores, ou seja, a complexidade é:

$$\mathcal{O}(1)$$

(Nota: na prática libertamos a memória que estava a meio das tangentes fazendo a complexidade linear mas poderiamos ter guardado num conjunto de poligonos a libertar, tornando a complexidade constante e no final teriamos de libertar, não mudando a complexidade total)

4.6 Calcular invólucro

$$\mathcal{T}(n) = \begin{cases} \mathcal{O}(1) & n = 0 \\ \mathcal{O}(1) & n = 1 \\ \mathcal{O}(n) + 2 * \mathcal{T}(n/2) & n > 1 \end{cases}$$

$$\mathcal{T}(n) = \sum_{i=0}^{\lfloor \log_2 n \rfloor} 2^i * \mathcal{O}(n/2^i) = \sum_{i=0}^{\lfloor \log_2 n \rfloor} \mathcal{O}(n) = (\lfloor \log_2 n \rfloor + 1) * \mathcal{O}(n) = \mathcal{O}(n \log n)$$

$$\mathcal{O}(n \log n)$$

5 Conclusão

Com o desenvolvimento deste algoritmo, achamos uma boa ideia para questões de debugging e apresentação criarmos uma ferramenta gráfica, *visualize* para introdução de pontos e *animate* para visualizar as execuções do programa desenvolvido, passo a passo.
Constatamos que a complexidade referida no documento partilhado pelo professor (Computational geometry in C) referia erradamente a complexidade deste algoritmo, sendo não $\mathcal{O}(n^2)$, mas sim $\mathcal{O}(n \log n)$ como mostramos neste trabalho. Por nossa surpresa na prática acaba mais rápido que o previsto.
Este algoritmo tem uma grande vantagem relativamente aos outros, porque é facilmente paralelizável, assim atingindo a complexidade $\mathcal{O}(n)$, porque segue a técnica dividir e conquistar.

- O documento estava errado da complexidade.
- A parte de encontrar os limites pode ser mais eficiente, mas a complexidade não mudaria, porque temos de ordenar.
- Paralelismo, to be done
- ...