6. Algoritmo Quickhull

No livro de Mulmuley [11] geometria computacional é tratada como o estudo de problemas de busca e ordenação em dimensões maiores. Em um certo sentido, esses problemas em dimensões maiores, são generalizações de problemas de ordenação e busca em uma lista de elementos de um universo ordenado (versão unidimensional de ordenação e busca). Deste ponto de vista não é nenhuma surpresa o fato de que algoritmos de ordenação e busca têm sido uma constante fonte de inspiração para desenvolvimento de algoritmos em geometria computacional. O algoritmo que veremos nesta seção, com algumas variações, foi proposto independentemente por várias pessoas quase ao mesmo tempo (cf. Eddy [5], Bykat [1], Green & Silverman [8]). Devido a semelhança com o algoritmo `Quicksort` o algoritmo que veremos foi batizado de `Quickhull` por Preparata & Shamos (cf. Seção 3 do Capítulo 3 de [14]).

Observação. O material desta seção foi extraído do Capítulo 3 de [6], Capítulo 3 de [12] e Capítulo 3 de [14].

6.1. **Descrição do algoritmo: pré-processamento.** A idéia básica por trás do algoritmo é: para o "maioria" dos conjuntos de pontos é mais fácil descartar muitos pontos que estão no interior do fecho convexo e concentrar o trabalho nos pontos que estão próximos da fronteira.

Antes de aplicarmos o algoritmo a um conjunto S de pontos dados o nosso primeiro passo será fazermos um certo pré-processamento (pelo menos nesta versão do algoritmo) e encontrar os pontos extremos nas direções: mais alta, mais baixa, mais à esquerda e mais à direita. (Em caso de empate escolha, por exemplo, o pontos com a menor x-coordenadas ou y-coordenadas, dependendo do caso.) Ligue esses quatro pontos formando um quadrilátero. Note que os pontos no interior do quadrilátero são dispensáveis para efeito de construirmos o fecho convexo. (Veja Figura 5.) Agora nosso problema se reduz a quatro subproblemas independentes: encontrar o fecho convexo dos pontos contidos nas quatro regiões triangulares exteriores ao quadrilátero (concatenando as quatro cadeias correspondentes a essas regiões obteremos o fecho convexo do conjunto de pontos S).

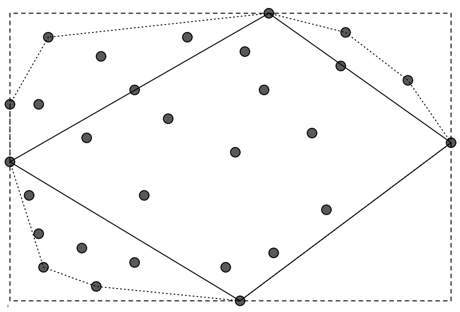

Figura 5. `Quickhull` descarta os pontos que estão no quadrilátero formado pelos 4 pontos extremos.

6.2. **Descrição do Algoritmo `Quickhull`.** O Algoritmo `Quickhull` resolve, recursivamente, cada um dos quatro subproblemas obtidos pelo pré-processamento. Chamemos de S o conjunto de pontos de um desses subproblemas e sejam a e b os pontos extremos de S que formam a base do triângulo. O algoritmo encontra um ponto extremo c de S, descarta pontos que estão no interior do triângulo formado pelos pontos a, b e c e particiona o problema em dois subproblemas idênticos menores.

Um pouco mais precisamente, em cada etapa do algoritmo nós temos um segmento ab e um conjunto de pontos S contidos em uma mesmo lado em relação à reta L que passa pelos pontos a e b e contidos na faixa formada pela reta L_a que passa pelo ponto a e é perpendicular ao segmento ab e pela reta L_b que passa pelo ponto b e é perpendicular ao segmento ab. Inicialmente, o algoritmo determina um ponto c que está o mais distante possível do segmento ab e cujo ângulo $\angle cab$ seja máximo. (Como podemos encontrar este ponto?) Note que este ponto é um ponto extremo de $\text{conv}(S)$. De fato, ele é um ponto extremo na direção da reta ortogonal ao segmento ab: os ponto de S estão de um mesmo lado da reta paralela a reta L que passa por c, e além disso c é o ponto mais à esquerda ou à direita nesta reta (dependendo do caso—direita e esquerda é em relação as x-coordenadas dos pontos), logo c não pode ser escrito como combinação convexa de pontos em S. (Veja Figura 6.)

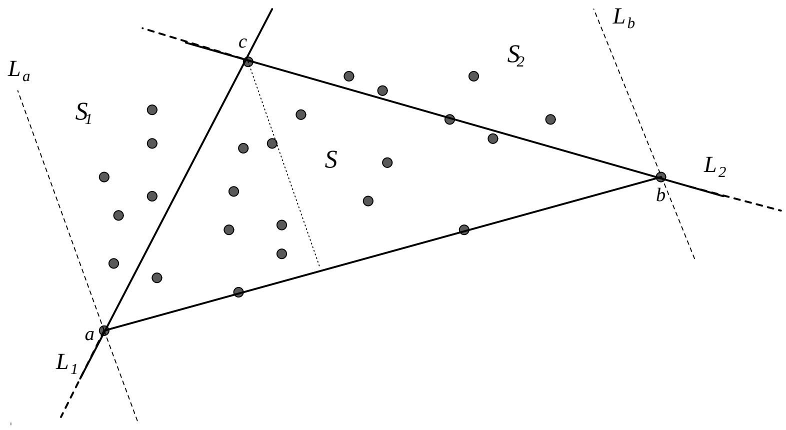

FIGURA 6. `Quickhull` descarta os pontos que estão no triângulo formado pelos pontos a, b e c.

O próximo passo do algoritmo é traçar retas L_1 passando por ac e L_2 passando por bc. Seja S_1 o conjunto de pontos de S que estão do lado oposto ao ponto b em relação à reta L_1 e seja S_2 o conjunto de pontos de S que estão do lado oposto ao ponto a em relação à reta L_2. Não existem pontos que estão simultaneamente em S_1 e S_2 ($S_1 \cap S_2 = \emptyset$) e os pontos de S que não estão nem em S_1 nem em S_2 são pontos que pertencem ao triângulo formado pelos pontos a, b e c. Agora o algoritmo, recursivamente, repete o mesmo procedimento com o segmento ac e o conjunto de pontos S_1 no lugar do segmento ab e o conjunto de pontos S. Analogamente, o mesmo procedimento é repetido para o segmento bc e conjunto de pontos S_2.

Algoritmo `Quickhull` (a, b, S)
`Entrada:` dois pontos distintos a e b e um conjunto de pontos S tal que todo
ponto de S está de um mesmo lado em relação à
reta que passa por a e b.
`Saída:` o fecho convexo de S.
if $S = \emptyset$ **then return** ab
else
$c \leftarrow$ ponto com distância máxima do segmento ab.
$S_1 \leftarrow$ pontos do lado oposto ao ponto b em relaçao à
reta L_2 passando pelo segmento ac.
$S_2 \leftarrow$ pontos do lado oposto ao ponto a em relaçao à
reta L_1 passando pelo segmento bc.
return `Quickhull`(a, c, S_1) concatenado com `Quickhull`(c, b, S_2).

A descrição completa do procedimento consiste do pré-processamento seguido de quatro chamadas ao Algoritmo `Quickhull`.

Observação. As operação primitivas usadas pelo algoritmo são `Left`, `Lefton` e `Area2`. Todas essas primitivas requerem um número pequeno de multiplicações, adições e subtrações (operações elementares).

6.3. **Análise do algoritmo.** Encontrar os quatro pontos extremos que formam o quadrilátero inicial durante o pré-processamento custa $O(n)$ operações elementares, onde n é o número de pontos.

A complexidade do Algoritmo `Quickhull` depende do número de pontos no conjunto S_1 e S_2. Seja $\alpha := |S_1|$ e $\beta := |S_2|$. Se denotaremos por $T(n)$ a função complexidade de tempo do Algoritmo `Quickhull` para encontrar o fecho convexo de um conjunto S com n pontos teremos que

$$T(n) \leq T(\alpha) + T(\beta) + cn,$$

onde c é uma constante universal. O termo cn na desigualdade acima corresponde ao número de operações gastas pelo algoritmo para encontrarmos o ponto c e construirmos os conjunto S_1 e S_2. Note que $\alpha + \beta \leq n$.

Quando $\alpha + \beta = n$ e cada subproblema tem aproximadamente o mesmo tamanho, por exemplo, $\alpha = \lfloor n/2 \rfloor$ e $\beta = \lceil n/2 \rceil$, teremos que

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn = O(n \log n).$$

Logo $T(n) = O(n \log n)$ é o melhor que pode ocorrer se tivermos $\alpha + \beta = n$.

O `Quickhull`, a exemplo do que ocorre com o `Quicksort`, é um algoritmo fácil de implementar que funciona bem quando os pontos de S são escolhidos aleatoriamente no plano. Entretanto, no pior caso, a sua complexidade de tempo é $O(n^2)$. Isso ocorre quando, $\alpha + \beta = n$ e (digamos) $\alpha = 1$ e $\beta = n - 1$. Nesse caso a nossa recorrência toma a forma:

$$\begin{aligned} T(n) &\leq T(n-1) + cn = T(n-2) + c(n-1) + cn \\ &= T(n-3) + c(n-2) + c(n-1) + cn = \cdots \end{aligned}$$

Repetindo a expansão nos chegaremos a

$$T(n) \leq c + c2 + \cdots + c(n-2) + c(n-1) + cn = O(n^2).$$

Logo, o `Quickhull` é um algoritmo que na prática é geralmente rápido, mas seu pior caso é quadrático (fica como exercício encontrar um conjunto de pontos genéricos onde o desempenho do `Quickhull` é realmente quadrático—por genérico queremos dizer uma construção de uma instância que 'funcione' para um valor de n arbitrariamente grande).