9. Um algoritmo de divisão-e-conquista

Muitos algoritmos têm a seguinte estrutura recursiva: para resolver um determinado problema eles fazem uma ou mais chamadas recursivas para resolver o mesmo problema em instâncias menores. Esses algoritmos tipicamente seguem o seguinte método de divisão-e-conquista: quebram o problema em diversos subproblemas que são similares ao problema original mas de tamanhos menores; resolvem recursivamente esses subproblemas menores; e finalmente combinam as soluções desses problemas menores para obter uma solução do problema para a instância original. (Veja Capítulo 1, Seção 3 de [3].)

Já vimos esse método antes durante o curso—lembre-se do algoritmo de complexidade de tempo $O(n \log n)$ para o Problema do Par Mais Próximo, do algoritmo de complexidade de tempo $O(n^4)$ para triangularização de polígonos e do Algoritmo `QuickHull`. Um exemplo bem-conhecido de algoritmo que usa este método é o `MergeSort`.

Mais especificamente, algoritmos de divisão-e-conquista têm três passos em cada nível da recursão:

Dividir: O problema é dividido em subproblemas de tamanho menor.

Conquistar: Os subproblemas são resolvidos recursivamente. Entretanto, se algum subproblema é suficientemente pequeno ele é resolvido diretamente.

Combinar: As soluções dos subproblemas são combinadas para obtermos uma solução do problema original.

Nesta seção nos veremos uma algoritmo de divisão-e-conquista que encontra o fecho convexo de um conjunto de n pontos no plano em tempo $O(n \log n)$ (um algoritmo ótimo para o problema). A técnica de divisão-e-conquista é a única conhecida que ao ser estendida para o Problema do Fecho Convexo tri-dimensional atinge a mesma complexidade de tempo, i.e. $O(n \log n)$. Na realidade, a primeira vez que este método foi utilizado para resolver o Problema do Fecho Convexo foi em 1997 por Preparata & Hong [13] com o objetivo de encontrar o fecho convexo tri-dimensional. Por causa da sua similaridade com o `MergeSort`, o algoritmo que veremos é muitas vezes chamado de `MergeHull` (veja, por exemplo, [6], [10] e [14]).

Observação. O material desta seção foi extraído do Capítulo 1 de [3], Capítulo 4 de [4], Capítulo 3 de [6], Capítulo 8 de [10], Capítulo 3 de [12], e Capítulo 3 de [14].

9.1. A descrição do algoritmo.

Para simplicar a exposição assumiremos que qualquer três pontos são não-colineraes (posição geral) e qualquer dois pontos não têm a mesma x-coordenada. O algoritmo está descrito logo abaixo. Note que para que o fecho convexo seja contruído em tempo $O(n \log n)$ necessitamos que a fase `Combinar` seja feita em tempo $O(n)$.

Algoritmo `MergeHull`(S)
`Entrada:` Um conjunto S de pontos no plano.
`Saída:` O fecho convexo de S.

1. **if** $|S| \leq k_0$, onde k_0 é uma constante pequena, construa o fecho convexo diretamente por algum método e **return**.
2. (`Dividir`) Particione o conjunto S em conjuntos S_1 com $\lceil n/2 \rceil$ pontos e S_2 com $\lfloor n/2 \rfloor$ pontos.
3. (`Conquistar`) Recursivamente construa o fecho convexo de S_1 e S_2.
4. (`Combinar` ou `Merge`) Tendo os fechos convexos de S_1 e S_2 construa o fecho convexo de $S_1 \cup S_2$.

9.2. **Pré-processamento: 'sorting step'.** Para facilitar a fase `Conquistar`, antes de aplicarmos o algoritmo `MergeHull`, ordenaremos o conjunto de pontos dado em relação as suas x-coordenadas.

9.3. **Dividir.** O conjunto de pontos S será divido em dois conjunto em relação a mediana das x-coordenas. Graças ao pré-processamento este passo pode ser feito rapidamente. Em outras palavras, o conjunto S será divido com relação a uma reta vertical: o conjunto de pontos S_1 será formado por os pontos de S que têm x-coordenada (digamos) menor ou igual a mediana das x-coordenadas dos pontos e o conjunto S_2 será formado pelos demais pontos de S (pontos à direita de reta vértica passando pela mediana das x-coordenadas. Note esta escolha de conjuntos S_1 e S_2 juntamente com a nossa hipótese de que S não tem dois pontos com a mesma x-coordenada garante que os fechos convexos de S_1 e S_2 são disjuntos, i.e. $\mathrm{conv}(S_1) \cap \mathrm{conv}(S_2) = \emptyset$.

9.4. **Conquistar.** O passo conquistar corresponde as chamadas recursivas onde cada subproblema será resolvido independentemente. A base da recursão é atinginda quando o conjunto de pontos em questão tem no máximo k_0 pontos, onde k_0 é uma constante. Na base da recursão o fecho convexo desses no máximo k_0 pontos pode ser construído por qualquer algoritmo, mesmo um algoritmo quadrático. Por exemplo, se $k_0 = 3$ o fecho convexo é um triângulo pela nossa hipótese de posição geral.

9.5. **Combinar: 'the merging step'.** A maior parte do trabalho deste algoritmo de divisão-e-conquista está neste passo. Precisamos ser cuidadoso para realizarmos este passo em tempo linear: um algoritmo ingênuo pode atingir a complexidade de tempo $O(n^2)$ para este passo, o que não nos permitiria obter a complexidade de tempo de $O(n \log n)$ para todo o algoritmo.

O trabalho deste passo é encontrar duas retas tangentes a $\mathrm{conv}(S_1)$ e $\mathrm{conv}(S_2)$: uma tangenciando os fechos convexos 'por baixo' e a outra tangenciando 'por cima'. A partir destas tangentes é fácil construirmos $\mathrm{conv}(S_1 \cup S_2)$ em tempo linear. Discutiremos como encontrar a tangente inferior; a tangente superior pode ser encontrada de maneira análoga.

Denotaremos por u_i e v_j os vértices de $\mathrm{conv}(S_1)$ e $\mathrm{conv}(S_2)$, respectivamente. Seja $T = u_i v_j$ o segmento de reta que determina a tangente inferior. O problema é que não sabemos de antemão quais são os vértices u_i de $\mathrm{conv}(S_1)$ e v_j de $\mathrm{conv}(S_2)$ que são extremidades de T. (Observe que fizemos uma tarefa semelhante no algoritmo incremental visto na seção anterior, mas, daquela feita a tarefa era mais simples pois conheciamos umas das extremidades de T.) Uma busca ingênua pelas extremidades de T resultaria em complexidade quadrática para este passo.

A busca pelo segmento tangente T feita por Preparata & Hong [13] começa inicializando T como sendo o segmento formado pelo vértice v_i mais à direita de $\operatorname{conv}(S_1)$ e pelo vértice v_j mais à esquerda de $\operatorname{conv}(S_2)$, e 'deixar T cair', primeiro 'escorregando' de um lado e depois do outro, e assim sucessivamente até que as extremidades do segmento procurado sejam encontradas. (Veja Figura 12.)

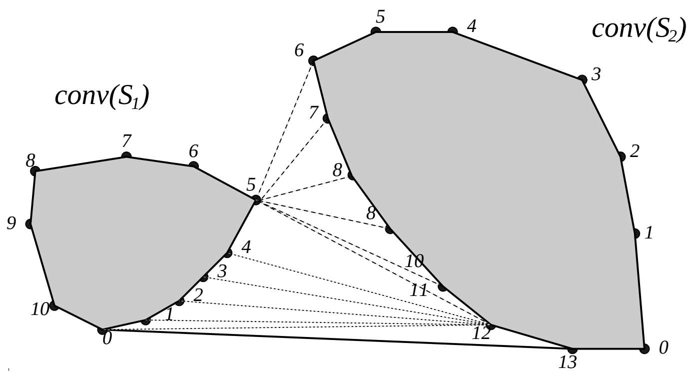

FIGURA 12. Procura pela tangente inferior: no início $T = (v_i, u_j) = (v_5, u_6)$ e a tangente procurada é (v_0, u_{13}).

Algoritmo `Tangente-Inferior`

`Entrada:` Dois fechos convexos $\operatorname{conv}(S_1)$ e $\operatorname{conv}(S_2)$, separados por uma reta vértical.

`Saída:` O segmento 'tangente' inferior a $\operatorname{conv}(S_1)$ e $\operatorname{conv}(S_2)$.

1. $v_i \leftarrow$ vértice mais à direita de $\operatorname{conv}(S_1)$.
2. $u_j \leftarrow$ vértice mais à esquerda de $\operatorname{conv}(S_2)$.
3. **while** $T = v_i v_j$ não é a tangente inferior de $\operatorname{conv}(S_1)$ e $\operatorname{conv}(S_2)$ **do**
4.   **while** T não é tangente inferior de $\operatorname{conv}(S_1)$ **do**
5. $j \leftarrow j + 1$
6.   **while** T não é tangente inferior de $\operatorname{conv}(S_2)$ **do**
7. $i \leftarrow i - 1$
8. **return** T.

No algoritmo acima, a frase '$T = v_i u_j$ não é tangente inferior de $\operatorname{conv}(S_1)$' é equivalente a dizermos: v_{i-1} está à direita de (v_i, u_j). Analogamente, a frase '$T = v_i u_j$ não é tangente inferior de $\operatorname{conv}(S_2)$' é equivalente a dizermos: u_{i+1} está à direita de (v_i, u_j). Da maneira semelhante ao que ocorreu com os algoritmos anteriores temos que a primitiva usada pelo Algoritmo `Tangente-Inferior` é `Lefton`.

Na Figura 12, inicialmente $T = v_5 u_6$ e o **while** da Linha 4 do algoritmo é executado até o momento em que u_{13} está à esquerda de $T = v_5 u_{12}$. Analogamente, o **while** da Linha 6 é executado até que v_{10} está à esquerda de $T = v_0 u_{12}$. Na próxima iteração do **while** da Linha 3 o segmento tangente $v_0 u_{13}$ é encontrado.

9.6. **Corretude do algoritmo.** Quando o **while** da Linha 3 do algoritmo pára temos que (a reta determinada por) T é tangente a ambos $\mathrm{conv}(S_1)$ e $\mathrm{conv}(S_2)$. Entretanto, precisamos verificar que:

1. O **while** da Linha 3 termina: a princípio poderia ocorrer que ao atingirmos a condição de tangência da Linha 6 tivessemos destruído a condição de tangência alcançada pela Linha 4.
2. Existem quatro tangentes a $\mathrm{conv}(S_1)$ e $\mathrm{conv}(S_2)$ (veja Figura 13), e o algoritmo deveria encontrar somente uma delas: a tangente inferior de ambos $\mathrm{conv}(S_1)$ e $\mathrm{conv}(S_2)$.

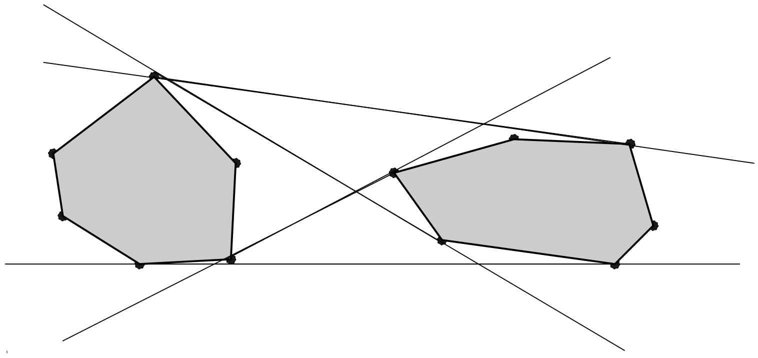

FIGURA 13. As quatro tangentes a $\mathrm{conv}(S_1)$ e $\mathrm{conv}(S_2)$.

O lemas abaixo serão usados para mostrar que os comandos das Linhas 5 e 7 do algoritmo não podem ser executados infinitamente, o que implica que o algoritmo sempre pára.

Lema 3. *Seja $T = vu$ a tangente inferior de conv(S_1) e conv(S_2). Então v (u, respectivamente) pertence à cadeia fechada que vai, no sentido anti-horário, do vértice mais à esquerda até o vértice mais à direita de conv(S_1) (conv(S_2), respectivamente).*

Prova. Seja L uma reta vertical separando $\mathrm{conv}(S_1)$ e $\mathrm{conv}(S_2)$. Se $\mathrm{conv}(S_2)$ está muito 'acima' de $\mathrm{conv}(S_1)$, então T se aproxima de ser paralelo com L e v_i se aproxima do vértice mais à direita de $\mathrm{conv}(S_1)$. Analogamente, se $\mathrm{conv}(S_2)$ está muito 'abaixo' de $\mathrm{conv}(S_1)$, então v_i se aproxima do vértice mais à esquerda de $\mathrm{conv}(S_1)$. O vértice v está entre estes extremos na cadeia inferior de $\mathrm{conv}(S_1)$. ∎

Lema 4. *(Invariante) Durante toda a execução do Algoritmo* **`Tangente-Inferior`** *temos que T tem intersecção vazia com o interior de conv(S_1) e conv(S_2) (i.e. $T \cap \mathrm{conv}(S_i) \subseteq \partial(\mathrm{conv}(S_i))$ para $i = 1, 2$).*

Prova. A prova é por indução no número de alterações de T feitas pelo Algoritmo `Tangente-Inferior`. A afirmação do Lema é verdadeira no começo do algoritmo.

Suponha que o Lema é verdade após alguns passos e que o algoritmo está prestes a decrementar a variável i; isto só é feito se T não é tangente inferior de $\mathrm{conv}(S_1)$. A nova tangente $T' = v_{i-1}u_j$ intersecta o interior de $\mathrm{conv}(S_1)$ se u_j está à esquerda de (v_{i-1}, v_i). (Veja Figura 14.) Mas

u_j não poderia também estar à esquerda de (v_i, v_{i+1}) pois senão T estaria intersectando o interior de conv(S_1). Logo, se o próxima atualização causasse a intersecção de T com o interior de conv(S_1) teriamos então que (a reta definida por) T seria tangente a conv(S_1) em v_i e portanto o algoritmo nunca teria incrementado a variável i. ∎

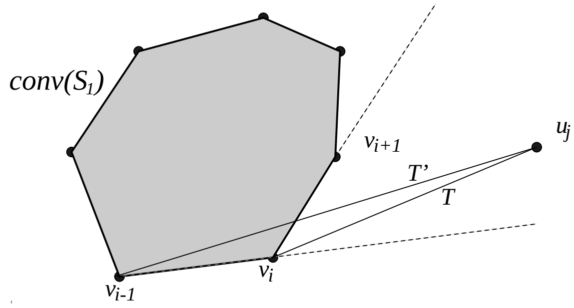

FIGURA 14. O vértice u_j de conv(S_2) está à esquerda de (v_{i-1}, v_i).

Como T não intersecta o interior de conv(S_1), e u_j está à direita da reta vertical L que separa conv(S_1) e conv(S_2), temos que a variavél i (Linha 7) não pode ser decrementada indefinidamente pois v_i não pode passar (andando no sentido horário) do vértices mais à esquerda de conv(S_1). De maneira análoga, a variavél j (Linha 5) não pode ser incrementada indefinidamente. Portanto, ambos os comandos **while** das Linha 4 e 6 devem parar e também o comando **while** da Linha 3 (pois em cada iteração do laço das Linhas 3–7 ou a variável i e decrementada ou a variável j e incrementada). Como o algoritmo pára então, pela condição de tangência (Linha 3) o algoritmo deve devolver a tangente inferior.

9.7. **Análise do algoritmo.** A complexidade de tempo do pré-processamento (ordenação dos pontos) que é $O(n \log n)$. No passo `Combinar` os laços são executados $O(n)$ vezes. Se usarmos uma lista circular duplamente ligada para representar os fechos convexos então, dadas as tangentes inferior e superior o novo fecho convexo pode ser construído em tempo $O(n)$. Portanto, a complexidade total do passo `Combinar` é $O(n)$. Se $T(n)$ é a função complexidade de tempo do Algoritmo `MergeHull`, sem contar o custo do pré-processamento, temos que

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

onde c é uma constante. Logo, a complexidade de tempo do Algoritmo `MergeHull` é $O(n \log n)$.