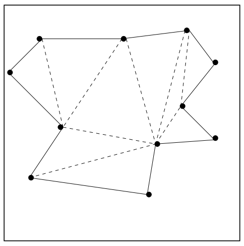
MAC 5747 GEOMETRIA COMPUTACIONAL

ALGORITMOS PARA PARTIÇÃO DE POLÍGONOS

1. Introdução



O interesse aqui é decompormos um certo 'domínio complexo' em uma coleção de objetos 'simples'. A região mais simples na qual podemos decompor um objeto planar é um triângulo (um tetraedro em 3-d e um 'simplex' em geral). Dado um polígono P queremos adicionar o maior número possível de diagonais (que não se cruzem) a P de tal forma que o interior do polígono fique particionado em triângulos.

Nesta parte da disciplina veremos um algoritmo que triangulariza um polígono dado em tempo $O(n \log n)$, onde n é o número de vértices do polígono. Por um longo tempo um dos principais problemas em aberto em geometria computacional foi a exsitência ou não de um algoritmo para triangularização de complexidade de tempo $o(n \log n)$. Em 1988 Tarjan e Van Wyk [25], quebraram a barreira do $O(n \log n)$ e apresentaram um algoritmo de complexidade de tempo $O(n \log \log n)$ — este algoritmo foi mais tarde simplificado por Kirkpatrick, Klawe e Tarjan [15]. Clarkson, Tarjan e Van Wyk [4], Devillers [8] e Seidel [23] desenvolveram algoritmos probabilísticos de complexidade de tempo $O(n \log^* n)$, onde $\log^* n$ representa o 'logaritmo iterado' de n. A questão sobre a existência ou não de um algoritmo linear para triangularização ficou em aberto até 1990, quando Chazelle [2] apresentou um algoritmo determinístico de complexidade de tempo linear para o problema.

Nesta seção, além de algoritmos para triangularização de polígonos, estudaremos algoritmos para particionar um polígono em polígonos monótonos e polígonos convexos.

Na Seção 2, veremos como determinar se um segmento conectando dois vértices é ou não uma diagonal de um dado polígono. Na Seção 3, apresentaremos dois algoritmos ingênuos para triangularização de polígonos — ambos são baseados em teoremas já estudados nesta disciplina. Estudaremos, na Seção 4, um algoritmo de complexidade de tempo linear para triangularizar polígonos 'monótonos', devido Garey, Johnson, Preparata e Tarjan [10]. Na Seção 5, apresentaremos o algoritmo por linha-de-varredure de Lee e Preparata [17] que particiona um polígono de n vértices em polígonos (partes) monótonos em tempo $O(n \log n)$ e espaço $O(n)$. Combinando os algoritmos das Seções 4 e 5 obteremos um algoritmo para triangularização de polígono que tem complexidade de tempo $O(n \log n)$ e complexidade de espaço linear.

Date: 3 de maio de 2000.

A Seção 6 trata do problema de particionar um polígono em partes convexas. Finalmente, na Seção 7, temos uma série de exercícios.

Observação. Os tópicos que cobriremos a seguir foram extraídos do Capítulo 3 de De Berg, Van Kreveld, Overmars e Schwarzkoft [6], Capítulo 5 e 7 de Laszlo [16], Capítulos 1 e 2 de O'Rourke[20], das notas de aula de Mount [18], Capítulo 7 de Preparata e Shamos [22], e Capítulo 5 de Resende e Stolfi [7].

2. Teste de diagonalidade

Seja $P = (v_0, v_1, \dots, v_{n-1})$ um polígono. Nesta seção veremos como testar se um segmento de reta é ou não uma diagonal de P. Faremos uso, além do conhecido predicado $\texttt{Left}(a, b, c)$, do predicado $\texttt{LeftOn}(a, b, c)$ que é equivalente a (ou abreviação de) $(\neg\texttt{Left}(b, a, c))$.

Recordemos que uma diagonal de P é um segmento de reta entre dois vértices distintos a e b que se vêem claramente, ou seja, $ab \subset P$ e $ab \cap \partial P = \{a, b\}$. Da definição de diagonal segue o seguinte lema.

Lema 1. *Seja $P = (v_0, v_1, \dots, v_{n-1})$ um polígono. Então $s = v_i, v_j, i \neq j$ é uma diagonal de P se e somente se*

(1) *para cada aresta a de P, que não é incidente a v_i ou v_j, temos que s e a não se intersectam ($s \cap a = \emptyset$); e*
(2) *$s \subset P$ em uma vizinhança de v_i ou de v_j.*

∎

Para testarmos se um segmento $s = v_i v_j$ satisfaz a condição (1) do lema basta aplicarmos o teste de intersecção entre s e no máximo $n - 4$ arestas de P. Estes testes de intersecção podem ser realizados em tempo $O(n)$.

Para determinarmos se $s = v_i v_j$ está no interior do polígono em uma vizinhança de, diagamos, v_i, temos que considerar dois casos.

Caso 1: v_i é vértice convexo. O vértice v_i é convexo se o predicado

$$\texttt{LeftOn}(v_{i-1}, v_i, v_{i+1})$$

é verdadeiro (a situação encontra-se ilustrada na Figura 1). Neste caso, o segmento $s = v_i v_j$ está no interior de P na vizinhança de v_i se e somente se ambos os predicados $\texttt{Left}(v_i, v_j, v_{i-1})$ e $\texttt{Left}(v_j, v_i, v_{i+1})$ são verdadeiros, ou seja,

$$\texttt{Left}(v_i, v_j, v_{i-1}) \wedge \texttt{Left}(v_j, v_i, v_{i+1})$$

é verdadeiro (verifique!).

Caso 2: v_i é vértice reflexo. O vértice v_i é reflexo se ele não for convexo (trocamos seis por meia dúzia?!?). Ou seja, o vértice v_i é reflexo se o predicado

$$\neg\texttt{LeftOn}(v_{i-1}, v_i, v_{i+1})$$

é verdadeiro (a situação encontra-se ilustrada na Figura 2). Neste caso, o segmento $s = v_i v_j$ está no interior de P na vizinhança de v_i se e somente se o predicado

$$\neg(\texttt{Left}(v_i, v_j, v_{i-1}) \wedge \texttt{Left}(v_j, v_i, v_{i+1})$$

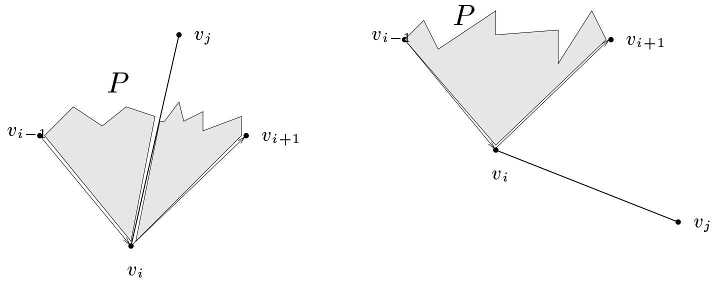


FIGURA 1. Possíveis situações quando v_i é convexo.

é verdadeiro (verifique!).

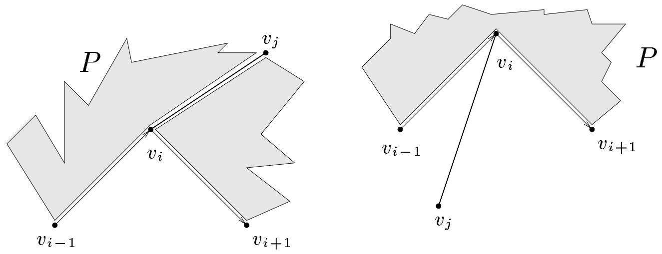


FIGURA 2. Possíveis situações quando v_i é reflexo.

Algoritmo `Diagonal`(P, v_i, v_j)
`Entrada:` um polígono $P = (v_0, \dots, v_{n-1})$ e vértices v_i e v_j, $v_i \neq v_j$.
`Saída:` TRUE se $s := v_i v_j$ é ume diagonal de P e FALSE caso contrário.
1. **for** toda aresta e de P não incidente a v_i ou v_j **do**
2.   **if** e e s se intersectam **then return** FALSE;
3. **if** v_i é convexo (`LeftOn`(v_{i-1}, v_i, v_{i+1})) **then**
4.   **return** `Left`$(v_i, v_j, v_{i-1}) \wedge$ `Left`(v_j, v_i, v_{i+1});
5. **else**
6.   **return** $\neg($`Left`$(v_i, v_j, v_{i-1}) \wedge$ `Left`$(v_j, v_i, v_{i+1}))$.

A complexidade de tempo do algoritmo é determinada pela Linha 1 que gasta tempo $O(n)$.

Teorema 2. *Seja P um polígono com n vértices e sejam u, v vértices de P. Então podemos decidir em tempo $O(n)$ se o segmento uv é um diagonal de P.* ∎

3. Dois algoritmos ingênuos

O primeiro algoritmo que veremos é baseado no Teorema da Triangularização (que vimos quando estudavamos o Teorema da Galeria da Arte).

Algoritmo `Ingênuo`(P)
`Entrada:` um polígono $P = (v_0, \dots, v_{n-1})$.
`Saída:` um triangularização de P.

if $n > 3$ **then**
    while não achou diagonal **do**
        Seja $v_i v_j$ um segmento candidato a diagonal.
        if `Diagonal`(P, v_i, v_j) **then**
            Achou diagonal;
        Trace a diagonal $v_i v_j$;
        Particione P em subpolígonos $P_1 = (v_0, \ldots, v_i, v_j, v_{j+1}, \ldots, v_{n-1})$
        e $P_2 = (v_i, v_{i+1}, \ldots, v_{j-1}, v_j)$.
        Ingênuo(P_1);
        Ingênuo(P_2);

A corretude do algoritmo segue imediatamente do Teorema da Triangularização de Polígonos. Existem $\binom{n}{2} = O(n^2)$ candidatas a diagonal e testar se um segmento é diagonal gasta tempo $O(n)$. Repetir estas $O(n^3)$ computações para cada uma das $n-3$ diagonais terá um custo de $O(n^4)$ operações elementares, que é a complexidade de tempo do algoritmo.

Podemos acelerar o algoritmo por um fator de n se explorarmos o Teorema das Duas Orelhas.

Algoritmo `Ototomia`(P)
`Entrada:` um polígono $P = (v_0, \ldots, v_{n-1})$.
`Saída:` um triangularização de P.
1. **while** $n > 3$ **do**
2.   Encontre v_i tal que $E := \triangle(v_{i-1}, v_i, v_{i+1})$ é uma orelha de P;
3.   Trace a diagonal $v_{i-1} v_{i+1}$;
4. $P \leftarrow P - E$; /* corta orelha */

Para encontrar, na Linha 2 do algoritmo, um vértice v_i que seja ponta de um orelha precisamos fazer no máximo n testes de diagonalidade (note que, devido ao Teorema das Duas Orelhas de Meister sabemos que qualquer polígono possui pelo menos duas orelhas). Logo, cada iteração da Linha 2 gasta tempo $O(n^2)$. Se o polígono for representado através de uma lista circular duplamente ligada, o tempo gasto com a Linha 3 é constante. Portanto, a complexidade de tempo do algoritmo é $O(n^3)$.

A complexidade de tempo do algoritmo de triangularização por corta-orelha ainda pode ser reduzida para $O(n^2)$. Já que um teste de diagonalidade gasta tempo $O(n)$ então, para obtermos um algoritmo quadrático, o teste de diagonalidade pode ser feito no máximo $O(n)$ vezes. A idéia chave para acelerar o algoritmo é que a remoção de um orelha não muda 'muito' o polígono, em particular, a remoção não altera o fato de muitos dos seus vértices serem ou não potenciais pontas de uma orelha. Isto sugere que façamos um fase de pré-processamento, onde testamos cada um dos vértices para descobrirmos quais são orelhas (ou seja, testamos se os seus vértices vizinhos são extremos de uma diagonal). Só este pré-processamento já gasta $O(n^2)$, mas, depois do pré-processamento, o tempo gasto será linear. Veja a lista de exercícios.

Note que estamos interpretando a tarefa 'triangularizar um polígono' como 'devolva, em uma ordem arbitrária, as diagonais que formam uma triangularização do polígono'. Frequentemente, uma saída mais estruturada é exigida, o que, em termos de complexidade assintótica de algoritmos, não é mais difícil (isto frequentemente só complica a descrição do algoritmo).

4. Triangularização de polígonos monótonos

Um polígono é chamada do *monótono em relação a uma reta* l se, para toda reta l' perpendicular a l, a intersecção do polígono com l' é conexa. Em outras palavras, a intersecção de l' e polígono deve ser um segmento de reta, um ponto, ou vazia. Um polígono que é monótono em relação ao eixo das ordenadas é chamado de y-monótono. A seguinte propriedade é característica de polígono y-monótonos: se percorrermos a cadeia direita (ou esquerda) formada pela fronteira ∂P do polígono, então nós sempre nos movemos para baixo ou horizontamente, mas nunca para cima.

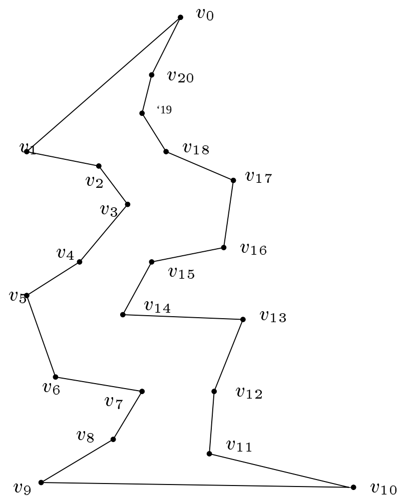


FIGURA 3. Polígono y-monotono.

Um curva poligononal ou cadeia (*chain*) é dita de *monótona em relação a uma reta* l se, para toda reta l' perpendicular a l, a intersecção entre a curva e l' é um ponto ou um segmento de reta. A fronteira de um polígono y-monótono é composta por duas curvas poligonais monótonas: a cadeia direita e a cadeia esquerda. Cada cadeia termina em um dos vértices mais baixo e em um vértices mais altas do polígono, e contém zero ou mais vértices entre estes extremos. A Figura 4 mostra alguns exemplos. A seguir nos concentraremos em polígonos y-monótonos

Um fato fundamental do ponto de vista de algoritmos envolvendo polígonos y-monótonos é que os seus vértices podem ser ordenados em relação às suas y-coordenadas em tempo linear. De fato, seja $P = (v_0, v_1, \ldots, v_{n-1})$ um polígono y-monótono. Suponha que P é representado através de uma lista circular duplamente ligada. Para obtermos uma ordenação $u_1, \ldots, u_n$ dos vértices de P em relação as suas y-coordenadas podemos proceder da seguinte maneira:

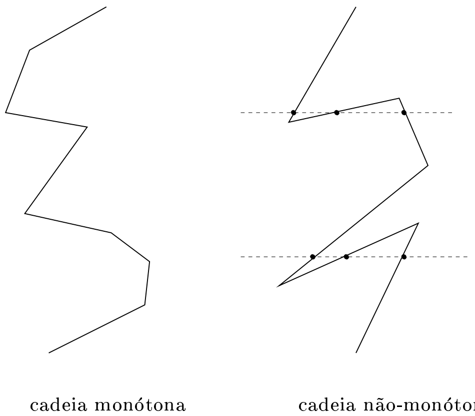


FIGURA 4. Cadeia y-monótona e não y-monótona.

1. Encontre um vértice mais alto e um vértices mais baixo de P. [Podemos executar este passo em tempo $O(n)$, simplesmente percorrendo a lista ligada.]
2. Particione ∂P em duas cadeias y-monótonas (a cadeia esquerda e a cadeia direita). [Este passo gasta tempo $O(1)$. É feito através de uma simples manipulação ed apontadores.]
3. Intercale (*merge*) as duas cadeias para formar uma lista dos vértices ordenados em relação as suas y-coordenadas. [O bem-conhecido algoritmo de intercalação gasta tempo $O(n)$.]

4.1. **Algoritmo.** O algoritmo que veremos triangulariza um polígono y-monótono em tempo linear. Seja $P = (v_0, \dots, v_{n-1})$ o polígono y-monótono e sejam $u_1, u_2, \dots, u_n$ os vértices de P listado em ordem decrescente de y-coordenadas. Para simplificar a exposição suporemos que P é *estritamente y-monótono*, ou seja, não existem dois vértices de P com uma mesma y-coordenada.

O algoritmo produz uma seqüência de polígonos

$$P = P_0, P_1, \dots, P_n = \emptyset.$$

O polígono P_i é obtido a partir de P_{i-1} após o algoritmo examinar o vértice u_i e, eventualmente, remover zero ou mais triângulos de P_{i-1}. O algoritmo termina após obtermos P_n o polígono vazio — a coleção de triângulos acumulados ao longo da execução do algoritmo representa a triangularização do polígono P dado.

O algoritmo mantém uma pilha S, no início da primeira iteração S contém u_1 na base e u_2 no topo. A pilha S conterá todos os vértices visitados que ainda não foram inteiramente processados pelo algoritmo. Determinados invariantes serão mantidos pelo algoritmo. Mais especificamente, se denotarmos por $s_1, \dots, s_t$ os vértices em S da base até o topo, as seguintes condições são satisfeitas no início da iteração i do algoritmo:

1. $s_1, s_2, \dots, s_t$ estão em ordem decrescente de y-coordenadas e incluem todos os vértices de P_{i-1} que estão abaixo de s_1 e acima de s_t;
2. $s_1, s_2, \dots, s_t$ são vértices consecutivos na cadeia esquerda ou direita de P_{i-1};

3. os vértices $s_2, s_3, \dots, s_{t-1}$ são vértices reflexos de P_{i-1} — por esta razão a subcadeia $s_1, \dots, s_t$ é chamada de *cadeia reflexa*, ou seja, a pilha mantém uma cadeia reflexa do polígono corrente.
4. o vertice u_i de P_{i-1}, que será visitado pelo algoritmo na presente iteração, satisfaz uma das seguintes condições:
   (a) u_i é adjacente a s_t, mas não é adjacente a s_1;
   (b) u_i é adjacente a s_1, mas não é adjacente a s_t;
   (c) u_i é adjacente a ambos s_1 e s_t.

Devido ao invariante 2 e 3 chamaremos a subcadeia formada por $s_1, \dots, s_t$ de *cadeia reflexa* corrente, ou seja, a pilha mantém uma cadeia reflexa do polígono corrente

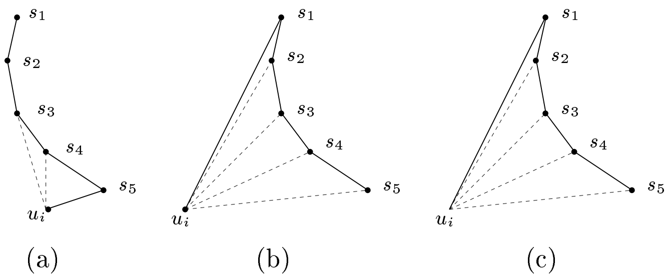


FIGURA 5. Ilustração dos três casos que podem ocorrer durante a triangularização de um polígono y-monótono: o vértice u_i sendo examinado é (a) adjacente a s_t mas não a s_1, (b) adjacente a s_1 mas não a s_t, ou (c) u_i é adjacente a ambos s_1 e s_t.

O processamento do vértice u_i depende de qual das condições 4(a), 4(b) ou 4(c) é satisfeita. A Figura 5 ilustra cada um dos casos. O algoritmo abaixo triangulariza um polígono y-monótono em tempo linear. Note que o algoritmo traça diagonais sem sequer verificar visibilidade clara, ou seja, nenhum teste de diagonalidade é executado (lembremos que testar se um segmento é diagonal de um polígono consome tempo $O(n)$). Durante o algoritmo a cadeia reflexa será mantida através de uma lista duplamente liga S com um apontador para o `topo` e outro para a `base`.

Algoritmo `Monótono`(P)
`Entrada:` um polígono monótono $P = (v_0, \dots, v_{n-1})$.
`Saída:` um triangularização de P.

1. Ordene os vértices $v_0, v_1, \dots, v_{n-1}$ em ordem decrescente de y-coordenadas. Sejam $u_1, \dots, u_n$ os vértices nessa ordenação.
   [Este passo gasta tempo $O(n)$, como vimos anteriormente.]
2. Inicialize a cadeia reflexa S com os dois vértices mais altos, ou seja $S \leftarrow (u_1, u_2)$.
3. **for** $i = 3, \dots, n$ **do**
   Sejam $s_1, s_2, \dots, s_t$ os vértices em S da `base` até o `topo` da pilha (s_1 é o vértice mais alto da cadeia reflexa e s_t o mais baixo).
   Caso 4(a): u_i é adjacente a s_t, mas não é adjacente a s_1.
   Enquanto $t > 1$ e o ângulo $\angle u_i s_t s_{t-1} > \pi$ trace a diagonal $u_i s_{t-1}$,

desempilhe s_t e decremente t. O algoritmo usa o fato de que $\angle u_i s_t s_{t-1} > \pi$ somente se: (1) u_i está à direita do segmento orientado $s_{t-1}s_t$ se u_i pertence à cadeia direita ($\texttt{Right}(s_{t-1}, s_t, u_i) = \neg\texttt{Lefton}(s_{t-1}, s_t, u_i)$); ou (2) u_i está à esquerda do segmento orientado $s_{t-1}s_t$ se u_i pertence à cadeia esquerda ($\texttt{Left}(s_{t-1}, s_t, u_i)$).

Caso 4(b): u_i é adjacente a s_1, mas não é adjacente a s_t.
Trace as diagonais $u_i s_2, u_i s_3, \ldots, u_i s_t$. Desempilhe todos os vértices de S e empilhe s_t seguido de u_i. (Note que o polígono determinado pelos vértices $u_i, s_1, s_2, \ldots, s_t$ possui uma única triangularização.)

Caso 4(c): u_i é adjacente a ambos s_1 e s_t.
Neste caso $u_i = u_n$ e o polígono P_{i-1}, determinado pelos vértices $u_i, s_1, s_2, \ldots, s_t$ possui um única triangularização. Trace as diagonais $u_i s_2$, $u_i s_3$, $\ldots$, $u_i s_{t-1}$ e pare (ou volte para o **for**, da na mesma).

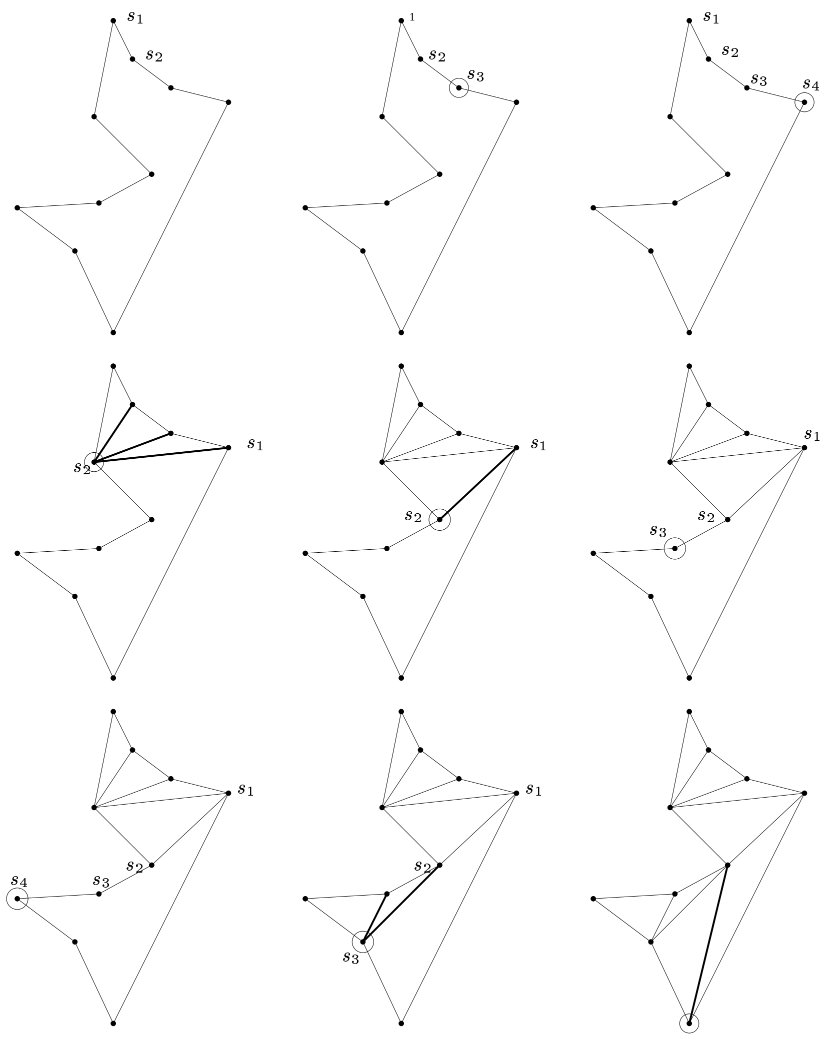


FIGURA 6. Triangularização de um polígono y-monótono. As diagonais encontradas em cada fase estão em negrito.

4.2. **Corretude do algoritmo.** A corretude do algoritmo segue da discussão que precedeu a sua apresentação.

4.3. **Análise do algoritmo.** Quanto tempo este algoritmo gasta para triangularizar um polígono y-monótono $P = (v_0, v_1, \ldots, v_{n-1})$? O laço **for** é executado $n - 3$ vezes, e uma execução pode gastar tempo linear, o que sugere que a complexidade de tempo do algoritmo é $O(n^2)$. Apesar disto ser verdade, podemos fazer melhor, basta sermos um pouco mais cuidadosos na análise. Notemos que em cada execução do laço **for** no máximo duas operações `empilha` são executadas. Assim, o número total de operações `empilha` executadas pelo algoritmo durante toda a sua execução é limitado por $2n - 4$, incluindo os dois **empilha** executados no Passo 2. Portanto, como o número de execuções da operação `desempilha` ao longo da execução do algoritmo não pode exceder o número de operações `empilha`, o tempo total gasto pelo laço **for** é $O(n)$.

Teorema 3. *Um polígono (estritamente) y-monótono com n vértices pode ser triangularizado em tempo linear.* ∎

Utilizaremos o Algoritmo `Monótono` para triangularizar polígonos y-monótonos como uma subrotina de um algoritmo que triangulariza um polígono de n vértices em tempo $O(n \log n)$. Para isto, na próxima seção, veremos um algoritmo que particiona um polígono em subpolígonos monótonos em tempo $O(n \log n)$ e espaço $O(n)$.

5. Partição em partes monótonas

O algoritmo que estudaremos nesta seção particiona um polígono dado em partes (polígonos) monótonos usando a técnica da linha de varredura (*sweepline technique*). Depois descreveremos como, utilizando este algoritmo, podemos obter um algoritmo de complexidade de tempo $O(n \log n)$ e de espaço $O(n)$ para triangularizar um polígono.

5.1. **Pontas interiores.** Definamos uma *ponta interior* de um polígono como sendo um vértice reflexo v que tem ambos os seus vizinhos v_- e v_+ acima ou abaixo de v (pelo menos um dos vizinhos deve estar extritamente acima ou abaixo), veja Figura 7.

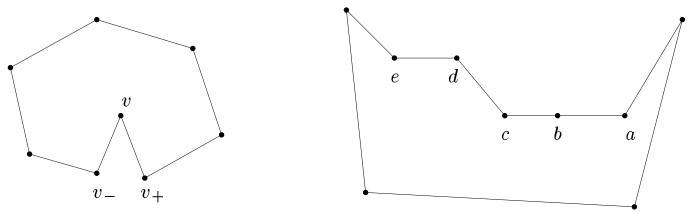


Figura 7. Ilustração de pontas interiores: (a) v é ponta interior (para cima); (b) a, c e e são pontas interiores (para baixo) e b e d não são pontas interiores.

Pontas interiores são fontes de não-monotonicidade local. O seguinte lema afirma que esta não-monicidade local é de fato a única razão de um polígono não ser y-monótono.

Lema 4. *Um polígono é y-monótono se ele não possui pontas interiores.*

Prova. Suponha que P é um polígono não y-monótono. Temos que mostrar que P contém uma ponta interior.

Como P é não y-monótono, existe uma reta horizontal l que intersecta P em mais de um componente conexo. Podemos escolher l tal que o componente mais à esquerda seja um segmento e não um simples ponto. Seja p o extremo esquerdo deste segmento e seja q o extremo direito. Começando em q nós seguiremos a fronteira de P de tal forma que o polígono está sempre a nossa esquerda. (Isto significa que nós 'subiremos' a partir de q.) Em algum ponto, digamos r, a fronteira irá intersectar l novamente. Se $r \neq p$, como na Figura 8(a), então o vértice mais alto que encontramos enquanto estavamos indo de q até r é necessariamente uma ponta interior (para cima).

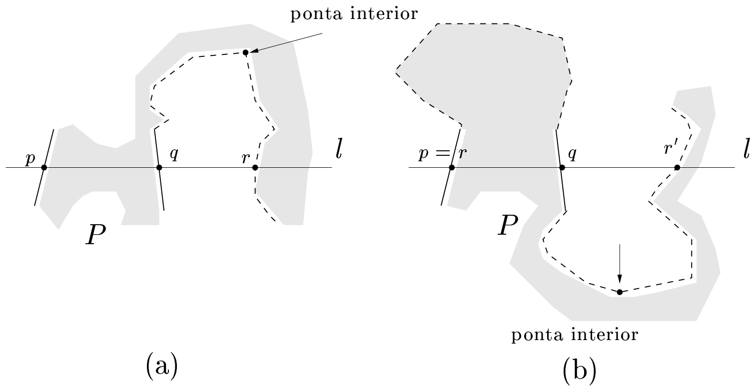


FIGURA 8. Dois casos na prova do Lema 4.

Se $r = p$, como na Figura 8(b), nós novamente seguiremos a fronteira de P a partir de q, mas desta vez na outra direção. Como anteriormente, a fronteira irá intersectar l novamente. Seja r' o ponto em que isto ocorre. Não podemos ter $r' = p$, pois isto significaria que a fronteira de P intersecta l apenas duas vezes, contrariando a hipótese de que l intersecta P em pelo menos dois componentes conexos. Assim, $r' \neq p$, implicando que o vértice mais baixo que encontramos no caminho de q até r' é necessariamente um ponta interior (para baixo). ∎

O Lema 4 implica que um polígono P terá sido particionado em partes y-monótonas assim que nos livramos das pontas interiores. Faremos isto adicionado diagonais que vão 'para cima' a partir de pontas interiores para cima e que vão 'para baixo' a partir de pontas interiores para baixo. Usaremos o conceito de trapezoidalização de um polígono para nos ajudar a compreender melhor o comportamento do algoritmo.

5.2. **Trapezoidalização horizontal.** Um *trapézio* é um quadrilátero que possui duas arestas paralelas. Uma *trapezoidalização horizontal* de um polígono P é obtida através do traçado de retas horinzontais passando por cada vértice do polígono. Mais precisamente, trace através de cada vértice v um segmento (aberto) maximal s tal que s está contido em P (ou seja,

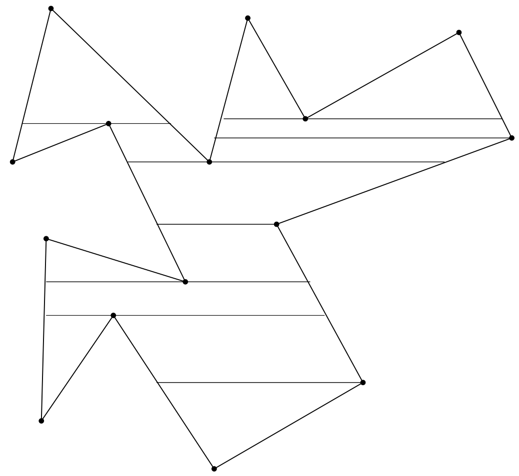


FIGURA 9. Um polígono particionado em trapézios horizontais.

$s \subset P$) e o único ponto comum entre a fronteira ∂P de P e s é v (ou seja, $s \cap \partial P = \{v\}$). A Figura 9 ilustra a trapezoidalização de um polígono.

Para simplificarmos a exposição consideraremos polígonos que não possuem dois vértices com uma mesma y-coordenada: não temos dois vértices sobre uma mesma reta horizontal.

Seja P um polígono. Em uma trapezoidalização horizontal de P, todo trapézio tem exatamente dois vértices de P em sua fronteira: um pertencente a sua aresta horizontal superior e outro na sua aresta horizontal inferior. Chamaremos estes vértices de *vertices de suporte* do trapézio.

A ligação entre partições em trapézios horizontais e polígonos y-monótonos é a seguinte: se um vértice de suporte está no interior de uma aresta de um trapézio então este vértice é uma ponta interior do polígono, e reciprocamente, se um vértice é ponta interior de um polígono então ele é um vértice de suporte interior a uma aresta de um trapézio. Assim, se ligarmos cada vértice de suporte interior a uma aresta de um trapézio ao vértice de suporte 'oposto' (para baixo se a ponta for para baixo e para cima se a ponta for para cima) esses segmentos serão diagonais que particionam P em partes y-monótonas. Isto segue do Lema 4, já que cada ponta interior será removido por estas diagonais: cada um desses vértices passará a ter um vizinho 'mais alto' e um 'mais baixo'. Na Figura 10 vemos o resultado desta idéia aplicada no polígono da Figura 9.

5.3. **O algoritmo de Lee e Preparata** [17]. Estamos agora preparados para descrevermos o algoritmo de Lee e Preparata [17], que utiliza a técnica da linha-de-varredura. Abaixo descrevemos quais são os pontos eventos, o funcionamente da fila de eventos, da estrutura da linha de varredura e como cada tipo de ponto evento deverá ser processado pelo algoritmo. Lembremos que, a fim de simplificarmos a exposição, estamos supondo que o polígono dado não contém dois vértices com uma mesma y-coordenada.

5.3.1. *Pontos eventos.* Os pontos eventos (*event points*) so algoritmo serão os vértices do polígono. A fila de pontos eventos *event-point schedule* será

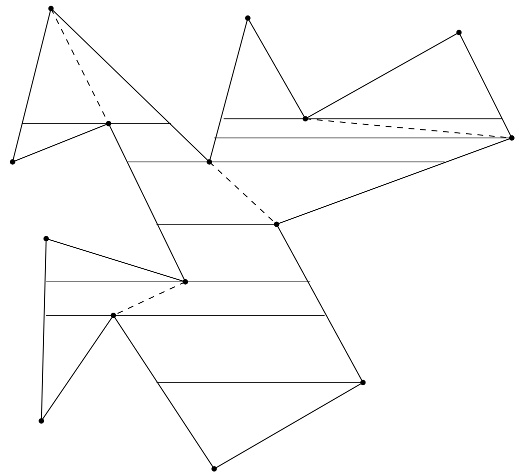


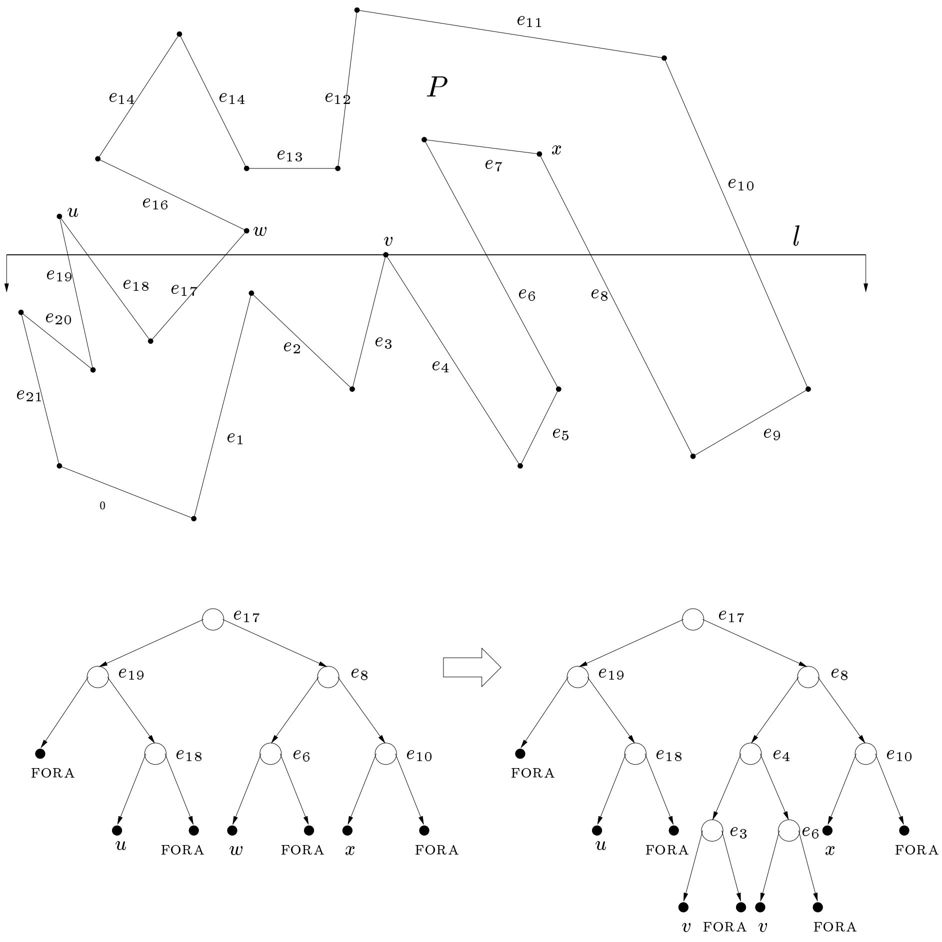
FIGURA 10. Polígono da Figura 9 particionado em cinco subpolígonos y-monótonos.

determinada estaticamente e consistirá de uma lista ordenada $\mathcal{Q}$ dos pontos do polígono em ordem decrescente da y-coordenada.

5.3.2. *Estrutura da linha-de-varredura.* Desta feita imaginaremos uma reta l varrendo o polígono de cima para baixo. A estrutura da linha de varredura (*sweepline structure*) $\mathcal{T}$ manterá as arestas do polígono que estão sendo intersectadas pela linha de varredura l em ordem das x-coordenadas dos pontos de intersecção (análogo ao que foi feito no algoritmo `Intersecção-de-Segmentos`). Além disso, a estrutura manterá os vértices do polígono que são vértices de suporte das arestas superiores dos *trapézios ativos*, ou seja, trapézios para os quais, até o momento, não se conhece o vértice de suporte da aresta inferior. Alguns desses vértices são pontas interiores para baixo que ainda precisam ser eliminadas, os outros são vértices que ajudarão a eliminar eventuais pontas interiores para cima.

A estrutura da linha de varredura deverá ser representada de uma maneira que permita operações de inserção, remoção e busca de segmentos e vértices em tempo $O(\log n)$ por operação. Para isso podemos usar como estrutura de dados uma árvore de busca balanceada como, por exemplo, árvore AVL, árvore 2-3, ... Na Figura 11(a) vemos o estado da estrutura da linha de varredura imediatamente antes da reta l alcançar o ponto evento v e na Figura 11(b) imediatamente após processar o ponto evento v.

5.3.3. *Processamento dos pontos eventos.* A atualização da estrutura da linha de varredura (*sweepline structure*) $\mathcal{T}$ depende do tipo do evento. Existem três tipos de pontos eventos. Suponha que v está entre as arestas a e b sobre l e que v é um vértice compartilhado pelas arestas c e d. Na representação das seqüências abaixo não estamos incluindo os vértices de suporte dos trapézios ativos — você pode imaginar que só estamos representando os nós internos da árvore balanceada que esta mantendo a estrutura da linha de varredura.



(a) (b)

Figura 11. Estrutura da linha de varredura. Os vértices u, w, x são vértices de suporte de trapézios ativos antes da linha de varredura alcançar o ponto evento v. Após o linha passar pelo ponto evento v, os vértices de suporte de trapézios ativos são u, v, x.

Caso 1: O segmento c está acima de l e d abaixo de l.
Remova c da estrutura $\mathcal{T}$ e insira d.

$$(\dots, a, c, b, \dots) \Longrightarrow (\dots, a, d, b, \dots)$$

Caso 2: Ambos c e d estão acima de l.
Remova c e d da estrutura $\mathcal{T}$.

$$(\dots, a, c, d, b, \dots) \Longrightarrow (\dots, a, b, \dots)$$

Caso 3: Ambos c e d estão abaixo de l. (Está é precisamente a situação do vértice v na Figura 11)
Insira c e d na estrutura $\mathcal{T}$.

$$(\dots, a, b, \dots) \Longrightarrow (\dots, a, c, d, b, \dots)$$

5.3.4. *Algoritmo de varredura.* O algoritmo, de uma maneira global, está descrito logo abaixo.

Algoritmo `FazMonótono`(P)
`Entrada:` um polígono $P = (v_0, \ldots, v_{n-1})$.
`Saída:` Uma partição de P em subpolígonos y-monotonos.

1. $\mathcal{T} \leftarrow \emptyset$. [Inicialização da *sweepline structure* $\mathcal{T}$.]
2. Ordene os vértices em ordem decrescente de y-coordenadas e coloque-os na lista ordenada (fila de pontos eventos) $\mathcal{Q}$
   [Este passo gasta tempo $O(n \log n)$.]
3. Para cada vértice v na lista ordenada $\mathcal{Q}$ faça
   Encontre a aresta a e b imediatamente a esquerda e direita de v e sejam c e d as arestas que compartilham v como extremidade.
   Caso 1: c está acima de l e d abaixo.
   Seja u o vértice de suporte da aresta superior do trapézio que tem o vértice v como vértice de suporte da aresta inferior.
   Remova c de $\mathcal{T}$ e insira d em seu lugar.
   Remova u da estrutura $\mathcal{T}$ e insira v em seu lugar.
   [Note que v foi inserido na estrutura da linha de varredura, pois agora v passa a ser o vértice de suporte da aresta superior de um trapézio.]
   Se u for uma ponta interior para baixo então trace a diagonal vu.
   [Eliminação de uma eventual ponta interior para baixo.]
   Caso 2: v é ponta interior para baixo (c e d estão acima de l.)
   Sejam u e w os vértices de suporte das arestas superiores dos dois trapézios que têm v como vértice de suporte da aresta inferior.
   [Note que um desses trapézios tem c como uma de suas arestas laterais e o outro tem d como uma de suas arestas laterais.]
   Remova de $\mathcal{T}$ os segmentos c e d.
   Remova de $\mathcal{T}$ os vértices u e w e insira o vértice v no lugar.
   [Note que v é uma ponta interior que passa a ser o vértice de suporte da aresta superior de um trapézio ativo — esta ponta interior será eliminada quando o vértice de suporte da aresta inferior do trapézio for encontrada.]
   Se u é ponta para baixo então trace a diagonal vu.
   Se w é ponta para baixo então trace a diagonal vw.
   [Eliminação de eventuais pontas interiores para baixo.]
   Caso 3: v é ponta interior para cima (c e d estão abaixo de l).
   Seja u o vértice de suporte da aresta superior do trapézio que tem v como vértice de suporte da aresta inferior.
   Insira os segmentos c e d em $\mathcal{T}$.
   Remova u de $\mathcal{T}$ e o ponto v.
   [Note que v passa a ser o vértice de suporte da aresta superior de dois trapézios.]
   Trace a diagonal entre vu.
   [Eliminação de uma ponta interior para cima.]

5.3.5. *Análise do algoritmo.* Podemos supor que o polígono P é dado através de um lista circular duplamente ligada do seus vértices, se P é dado de uma outra forma — através de uma lista de seus vértices no sentido anti-horário, por exemplo — então a primeira coisa que faremos é construir uma lista circular duplamente ligada dos vértices de P. As diagonais computadas durante a execução do algoritmo podem ser facilmente utilizadas para particionarmos o polígono em subpolígono em tempo constante, basta executarmos algumas simples manipulações de ponteiros.

Para cada vértice v precisamos encontrar a aresta imediatamente à esquerda e a aresta imediatamente à direita em l. Para fazer isso, sabendo as coordenadas dos pontos extremos de cada aresta, podemos calcular onde a linha de varredura corta a aresta e comparar com a y-coordenada do ponto. Precisamos fazer esta operação n vezes (n é o número de vértices do polígono) a um custo de $O(\log n)$ por vez (se usarmos uma árvore de busca balanceada para manter a *sweepline structure*).

O tempo gasto em cada um dos casos do passo 3 é $O(\log n)$ (inserções, buscas e remoções a um custo de $O(\log n)$ por operação). Isso nos da uma complexidade de tempo de $O(n \log n)$ para todo o algoritmo, pois o passo 3 será executado n vezes.

Teorema 5. *Um polígono com n vértices pode ser particionado em polígonos y-monótonos em tempo $O(n \log n)$ por um algoritmo que usa espaço $O(n)$.* ∎

6. Triangularização em tempo $O(n \log n)$

Utilizaremos um algoritmo para triangularização de polígonos monótonos como subrotina de um algoritmo para triangularizar polígonos arbitrários. A idéia é primeiro decompor o polígono dado em subpolígonos monótonos, usando para isto o Algoritmo `FazMonótono`. Depois basta aplicarmos o Algoritmo `Monótono` para triangularizar cada uma dos subpolígonos monótonos produzidos.

Quanto tempo é este algoritmo gasta? Decompor um polígono com n vértices gasta temp $O(n \log n)$, pelo Teorema 5. Numa segunda fase triangularizar cada uma das partes monótonas gasta tempo linear. Como a soma do número de vértices das partes monótonas é $O(n)$, a segunda fase gasta tempo total $O(n)$.

Teorema 6. *Um polígono com n vértices pode ser triangularizado em tempo $O(n \log n)$ por um algoritmo que usa espaço $O(n)$.* ∎

Vimos como triangularizar polígono 'simples', isto é, polígonos sem 'buracos' no meio. E se o polígono tiver buracos? Podemos também triangularizalo? A resposta é sim. De fato, o algoritmo que acabamos de descrever também pode ser usado para triangularizar polígono não-simples: em nenhum lugar do Algoritmo `FazMonótono` que particiona um polígono em partes monótonas, nós usamos o fato do polígono ser simples. A Figura 12 mostra uma triangularização de um polígono não-simples.

7. Partição em partes convexas

Além de algoritmos eficientes para particionar um polígono em triângulos, trapézios, partes monótonas, também é de interesse o desenvolvimento de algoritmos que particionem um polígono em polígonos convexos. Uma possível

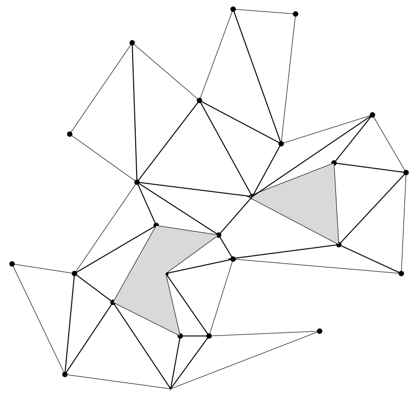
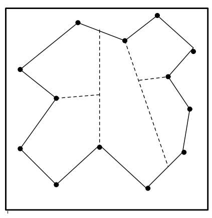


Figura 12. Traingularização de um polígono não simples.

motivação para particionarmos um polígono em partes (polígonos) convexos é reconhecimento de caracteres: um caractere pode ser representado como um polígono particionado em partes convexas.



A partição de polígonos em triângulos pode ser vista como um caso particular do problema de particionarmos um polígono em partes convexas.

Ao particionarmos um polígono em partes convexas estaremos interessados em *minimizar* o número de partes e construir esta partição o mais rápido possível.

Existem dois tipos de partições de um polígono P: (i) partições por diagonais, e (ii) partições por segmentos (os pontos extremos dos segmentos não são necessariamente vértices de P e podem não pertencer a ∂P).

Observação. O que será tratado nesta seçao foi extraído do Capítulo 1 de [19] e Capítulo 2 de [20].

7.1. **Limites.** Para analisarmos a qualidade (em termos de número de partes) de um partição é útil conhecermos limites inferiores e superiores para o menor número possível de partes em uma partição.

Teorema 7. *Seja Φ o menor número de partes convexas que um polígono com r vértices reflexos pode ser particionado. Então temos que*

$$\lceil r/2 \rceil + 1 \leq \Phi \leq r + 1.$$

Prova. A prova de que $\Phi \leq r + 1$ é algorítmica.
Algoritmo `Particiona` (P). `Entrada:` um polígono $P = (v_0, \ldots, v_{n-1})$.
`Saída:` Uma partição de P por segmentos em subpolígonos convexos.

1. Se P não tem vértices reflexos então **pare**. [Neste caso P já é convexo.]
2. Seja v um vértice reflexo de P.
3. Trace um segmento $s \subset P$ tendo v como uma das extremidades e a outra extremidade em ∂P que elimine este vértice reflexo.
4. Aplique o algoritmo recursivamente aos dois polígonos formados.

O número de partes convexas produzidas pelo algoritmo acima é no máximo $r+1$. (Veja Figura 13.)

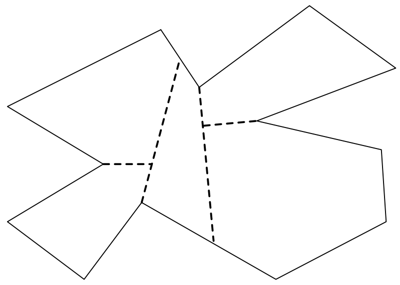


FIGURA 13. O algoritmo criou $r+1$ partes convexas ($r=4$).

Todos os ângulos reflexos precisam ser destruídos para produzirmos partes convexas. No máximo dois desses ângulos podem ser eliminados pela inclusão de um único segmento. Logo $\Phi \geq \lceil r/2 \rceil + 1$. (Veja Figura 14.) ∎

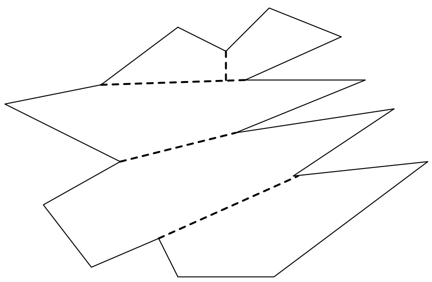


FIGURA 14. $\lceil r/2 \rceil + 1$ partes convexas ($r=7$).

7.2. **Algoritmo de Hertel e Mehlhorn.** Hertel e Mehlhorn [13] desenvolveram um algoritmo de aproximação de complexidade de tempo linear para particionar um polígono em partes convexas por diagonais.

Em uma partição convexa de um polígono por diagonais diremos que um diagonal d é *essencial para um vértice* v se a remoção de d cria uma parte que é não-convexa em v (v é um vértice reflexo e d é incidente a v).

O algoritmo de Hertel e Mehlhorn consiste em: construa uma triangularização de P; remova arbitrariamente diagonais não-essenciais até que todas as diagonais restantes sejam essenciais. Este algoritmo particiona um polígono em partes convexas em tempo linear (se usarmos o algoritmo de Chazelle [2] para triangularização de polígonos).

Lema 8. *Em uma partição convexa por diagonais existem no máximo duas diagonais essenciais para cada vértice reflexo.*

Prova. Seja v um vértice reflexo e v_+ e v_- os vértices adjacentes a ele. Existe no máximo uma diagonal essencial no semiplano H_+ à esquerda de (v, v_+); se existirem duas a que está mais perto de (v, v_+) pode ser removida sem que criemeos um vértice não-convexo em v. (Veja Figura 15.) Analogamente, existe no máximo uma diagonal essencial no semiplano H_- à esquerda de $(v_,v)$. Juntos esses semiplanos cobrem o ângulo interior formado pelas aresta vv_- e vv_+. Logo existem no máximo duas diagonais essencias para v. ∎

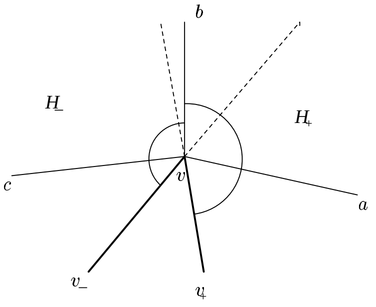


FIGURA 15. Diagonais essencias. A diagonal a é não-essencial pois b também está em H_+. Analogamente, c é não-essencial.

Teorema 9. *O algoritmo de Hertel-Mehlhorn fornece uma partição que nunca tem mais que quatro vezes o número de partes convexas de uma partição ótima por diagonais (i.e. uma partição por diagonais com o número mínimo de partes).*

Prova. Quando o algoritmo pára, toda diagonal é essencial para algum vértice (reflexo). Pelo Lema 8, cada vértice reflexo pode ser responsável por no máximo duas diagonais essenciais. Logo, o número de diagonais essenciais não pode ser maior do que $2r$, onde r é o número de vértices reflexos (pode ser menor se alguma diagonal for essencial para ambos os seus vértices extremos). Logo, o número M de partes convexas produzidas pelo algoritmo satisfaz $2r + 1 \geq M$. Como $\Phi \geq \lceil r/2 \rceil + 1$ pelo Teorema 7, então $4\Phi \geq 2r + 4 > 2r + 1 \geq M$. ∎

Problema 1. *(Melhor algoritmo de aproximação por diagonais) Encontre um algoritmo "rápido" que tem uma constante de optimalidade menor que 4. Por rápido queremos dizer $O(n \text{ polilog } n)$, onde polilog n é algum polinômio em $\log n$, como, por exemplo, $\log^3 n$.*

7.3. **Notas.** O primeiro algoritmo para encontrar uma partição ótima de um polígono em partes convexas é devido a Greene [11]: a complexidade de tempo deste algoritmo é $O(r^2 n^2) = O(n^4)$. Isto foi melhorado por Keil [14] que desenvolveu um algoritmo de complexidade de tempo $O(r^2 n \log n) = O(n^3 \log n)$. (Ambos utilizavam programação dinâmica.)

Se a partição pode ser formada por segmentos arbitrários o problema é mais difícil ainda (a partição pode usar segmentos cujos extremos não intersectam a fronteira do polígono — estes pontos extremos são chamados de *Steiner points*, veja Figura 16). Em 1980, em sua tese de doutorado, Chazelle [1] resolveu este problema propondo um algoritmo de complexidade de tempo $O(n + r^3) = O(n^3)$. (Veja também Chazelle e Dobkin [3].)

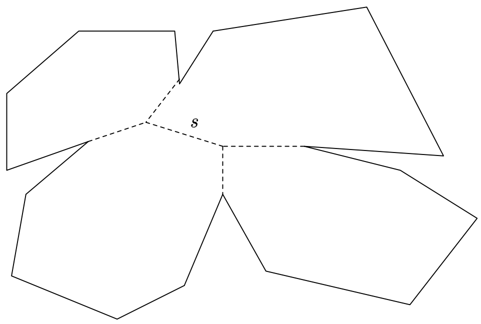


Figura 16. Uma partição convexa ótima. O segmento s não toca ∂P.

8. Exercícios

1. (Exercício 1.6.8.2 de [20]) Qual a complexidade de tempo do Algoritmo `Ototomia` quando restringimos a entrada a polígonos convexos?
2. (Ototomia em tempo $O(n^2)$) O algoritmo baseado em remoção de orelhas que foi visto em sala de aula tem complexidade de tempo $O(n^3)$. Nesta questão você desenvolverá um algoritmo de complexidade de tempo $O(n^2)$ que usa a mesma estrutura de dados e a função `Diagonal`.

   Uma chamada para a função `Diagonal` custa tempo $O(n)$. Logo, para obtermos um algoritmo de complexidade de tempo $O(n^2)$, a função `Diagonal` poderá ser chamada apenas $O(n)$ vezes. Imagine que você primeiro determina quais as potenciais diagonais de orelhas são de fato uma diagonal do polígono P, e marque o vértice v_i como 'ORELHA' se $v_{i-1}v_{i+1}$ é uma diagonal, caso contrário marque o vértice como 'NÃO-ORELHA'. Com isto, já gastamos $O(n^2)$ operações elementares.

   Seja P' o polígono de $n-1$ vértices resultante do polígono P após removermos uma de suas orelhas (digamos) $\triangle(v_{i-1}, v_i, v_{i+1})$. Responda as seguintes perguntas e justifique suas respostas:
   (a) É verdade que se o vértice v_j, $j \neq i$, está marcado como ORELHA em P então v_j continua sendo a ponta de uma orelha em P'?
   (b) Para quais valores de j a marca em v_j poderá ser alterada?
   (c) Quantas novas candidatas a diagonais precisam ser verificadas? (Ou seja, quantas novas chamadas da função `Diagonal` precisam ser feitas?)
   (d) Em quais condições um vértice marcado como NÃO-ORELHA em P deverá ser marcado como ORELHA em P'?
   (e) As marcas ORELHA e NÃO-ORELHA de P' herdadas por P podem ser atualizadas em tempo $O(n)$?

   Se a resposta para a última pergunta é sim então um algoritmo de triangularização de complexidade de tempo $O(n^2)$ pode ser obtido simplesmente mantendo-se as marcas ORELHA e NÃO-ORELHA dos vértices do polígono corrente e escolhendo-se, em cada iteração, uma orelha do polígono corrente para ser removida.

   Formule precisamente quais segmentos de P' precisam ser analisados para termos todas as orelhas de P' devidamente marcadas. Prove este lema, escreva o algoritmo resultante e a analise a sua complexidade de tempo.

3. Construa um polígono que é não-monótono em relação a qualquer reta.
4. (Exercício 1.6.8.2 de [20] — Pontas interiores) Contrua um polígono monótono que não seja estritamente monótono e que não tenha pontas interiores. (Logo, não é verdade que se um polígono não tem pontas interiores então ele é estritamente monótono.)
5. (Simulação: triangularização de polígonos monótonos) Simule (de alguma maneira clara) a execução do Algoritmo `Monótono` para triangularização de polígonos y-monótonos com o polígono da Figura 17.

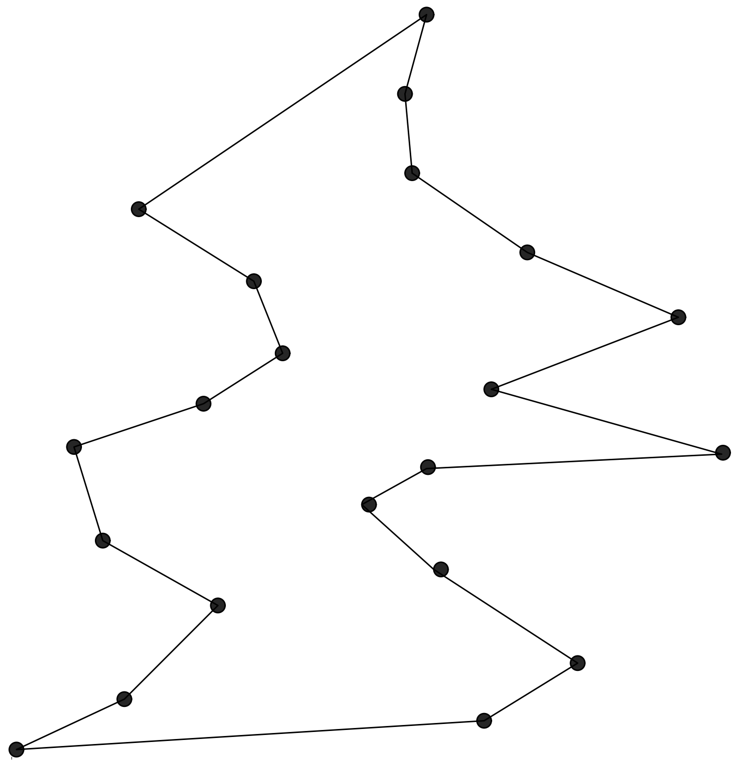


FIGURA 17. Um polígono monótono.

6. (Simulação: partição em polígonos monótonos) Simule (de alguma maneira clara) a execução do Algoritmo por linha-de-varredura `FazMonótono` que particiona um polígono em partes y-monótonos com o polígono da Figura 18.
7. (Exercício 1.6.8.2 de [20] — Monótono em relação a uma única direção) Um polígono pode ser monótono em relação a precisamente um única direção?
8. (Exercício 2.3.4.5 de [20] — Polígono $\Rightarrow$ quadriláteros convexos) Prove ou dê um contra-exemplo: Todo polígono com um número par de vértices pode ser particionado por diagonais em quadriláteros convexos.

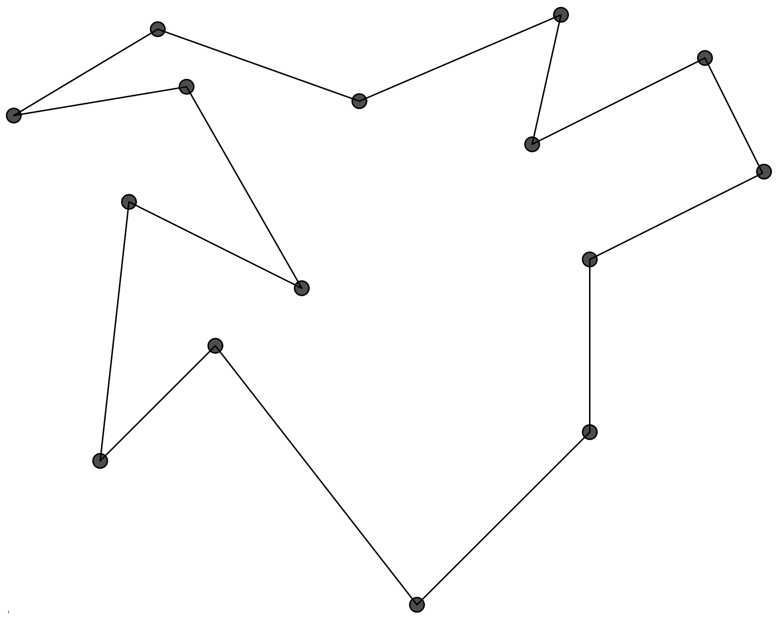


FIGURA 18. Um polígono para ser particionado em polígonos monótonos.

9. (Exercício 2.3.4.6 de [20] — Polígono ⇒ quadriláteros) Prove ou dê um contra-exemplo: Todo polígono com um número par de vértices pode ser particionado por diagonais em quadriláteros.
10. (Exercício 2.5.4.1 de [20] — Algoritmo de Hertel e Mehlhorn: pior caso em relação ao número de partes) Encontre um polígono genérico que pode levar ao pior caso do algoritmo de Hertel e Mehlhorn: existe uma triangularização e uma ordem para a remoção das diagonais não-essenciais que produz $2r$ partes convexas, onde r é o número de vértices reflexos do polígono).
11. (Exercício 2.5.4.2 de [20] — Algoritmo de Hertel e Mehlhorn: pior caso em relação ao ótimo) Encontre um polígono genérico que pode levar ao pior caso do algoritmo de Hertel e Mehlhorn em relação à partição ótima: o algorimo de Hertel e Mehlhorn produz $2r$ partes convexas, mas existe uma partição por diagonais com $\lceil r/2 \rceil + 1$ partes convexas, onde r é o número de vértices reflexos do polígono.
12. Seja P um polígono convexo cujos vértices estão em um vetor em ordem anti-horária. Descreva um algoritmo que decide se um dado ponto p pertence a P. A complexidade de tempo do seu algoritmo deve ser $O(\log n)$.
13. (*Minimum weight triangulation*) Este exercício mostra um exemplo de um algoritmo geométrico que faz decisões ótimas a cada passo, entretanto ele não fornece uma solução que é ótima globalmente. Dado um conjunto de pontos no plano, uma triangularização é um conjunto maximal de segmentos de reta ligando os pontos tal que dois a dois os segmentos não se intersectam "propriamente" (veja Figura 19). O peso de uma triangularização é a soma dos comprimentos dos segmentos.

Dado um conjunto de pontos no plano, o *Problema da Triangularização de Peso Mínimo* é o problema de encontrar, entre todas as triangularizações do conjunto de pontos dado, uma cujo peso seja mínimo.

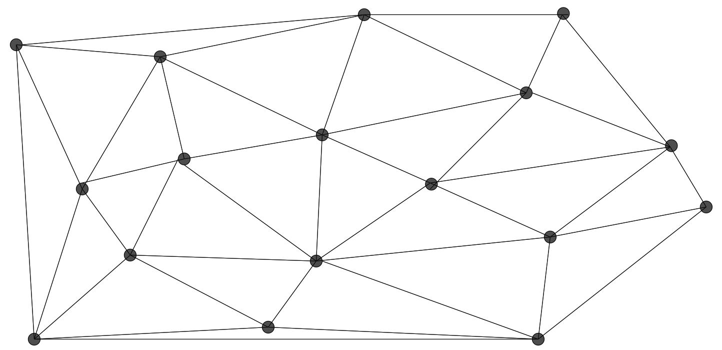


FIGURA 19. Uma triangularização de um conjunto de pontos.

[A complexidade do Problema da Triangularização de Peso Mínimo está aberta desde 1975 quando foi mencionado por Shamos e Hoey (cf. [24]). Não se conhece algoritmo polinomial para resolver este problema e também não se sabe se o problema é NP-difícil. O melhor algoritmo de tempo polinomial conhecido produz uma triangularização que, no pior caso é $O(\log n)$ vezes o peso de uma triangularização de peso mínimo (cf. Plaisted e Hong [21], veja também Heath e Pemmaraju [12]).]

Uma triangularização *gulosa* (*greedy*) de um conjunto de pontos é uma triangularização produzida pelo seguinte algoritmo:

Algoritmo Guloso(S)
`Entrada:` Um conjunto finito S de pontos no plano.
`Saída:` Uma triangularização de S.
$T \leftarrow \emptyset$;
$L \leftarrow$ o conjunto de todos os segmentos ligando pares de pontos de S;
while $L \neq \emptyset$ **do**
  Seja e um segmento em L de menor comprimento;
  Remova e de L;
  if (e não intersecta os segmentos em T) **then**
    Insira e em T;
endwhile

O Algoritmo Guloso escolhe uma aresta de comprimento mínimo em cada passo. Construa um instância S para o algoritmo tal que S tem 5 pontos e a triangularização gulosa de S não é uma triangularização de peso mínimo. Para isso:
(a) Forneça as coordenadas dos pontos;
(b) Mostre uma triangularização de peso mínimo de S;
(c) Mostre uma triangularização gulosa de S; e
(d) Calcule os pesos de ambas as triangularizações.

[Demonstra-se que a triangularização gulosa produz no pior caso uma triangularização que é $\Theta(n)$ vezes mais pesada que uma triangularização ótima.

O Capítulo 16 de [5] apresenta um algoritmo que encontra uma triangularização ótima de um polígono convexo através de programação dinâmica.

Sobre outros problemas de triangularização consulte o Capítulo 4, Seção 6 de [9].]

Referências

[1] B. Chazelle, *Computational geometry and convexity*, Ph.D. thesis, Yale University., 1980.

[2] ———, Triangulating a simple polygon in linear time, *Discrete and Computational Geometry* **6** (1991), 485–524.

[3] B. Chazelle e D.P. Dobkin, Optimal convex decompositions, *J. ACM* **34** (1987), 1–27.

[4] K.L. Clarkson, R.E. Tarjan e C.J. Van Wyk, A fast Las Vegas algorithm for triangulating a simple polygon, *Discrete and Computationa Geometry* **4** (1989), 423–432.

[5] T.H. Cormen, C.E. Leiserson e R.L. Rivest, *Introduction to algorithms*, The MIT Electrical Engineering and Computer Scienece Series, The MIT Press, MacGraw-Hill Book Company, 1990, QA758 C811i.

[6] M. de Berg, M. van Kreveld, M. Overmars e O. Schwarzkopf, *Computational geometry: Algorithms and applications*, Springer, 1997, Second Edition, 2000.

[7] P.J. de Resende e J. Stolfi, *Fundamentos de geometria computacional*, IX Escola de Computação, 1994.

[8] O. Devillers, Randomization yields simple $O(n\log^* n)$ algorithms for difficult $\Omega(n)$ problem, *International Journal of Computational Geometry and Applications* **2** (1992), 97–111.

[9] L.H. Figueiredo e P.C.P. Carvalho, *Introdução à geometria computacional*, $18^{\underline{o}}$ Colóquio Brasileiro de Matemática, IMPA, 1991, QA758 F475i.

[10] M.R. Garey, D.S. Johnson, F.P. Preparata e R.E. Tarjan, Triangulating a simple polygon, *Information and Processing Letters* **7** (1978), 175–179.

[11] D.H. Greene, The decomposition of polygons in convex parts, *Advances in Computing Research* (F.P. Preparata, ed.), JAI Press, 1983, pp. 235–259.

[12] L.S. Heath e S.V. Pemmaraju, New results for the minumum weight triangulation problem, *Algorithmica* **12** (1994), 533–552.

[13] S. Hertel e K. Mehlhorn, Fast triangulation of simple polygons, *Proc. 4th Internat. Conf. Found. Comput. Theory*, Lecture Notes in Computer Science, vol. 158, 1983, pp. 207–218.

[14] J.M. Keil, Decomposing a polygon into simpler components, *SIAM J. Comp.* **14** (1985), 799–817.

[15] D.G. Kirckpatrick, M.M. Klawe e R.E. Tarjan, Polygon triangulation in $O(n\log\log n)$ time with simple data structures, *Proceddings of the 6th Annual ACM Symposium on Computational Geometry*, 1990, pp. 34–43.

[16] M.J. Laszlo, *Computational geometry and computer graphics in C++*, Prentice Hall, Upper Saddle River, NJ, 1996.

[17] D.T. Lee e F.P. Preparata, Location of a point in a planar subdivision and its applications, *SIAM Journal on Computing* **6** (1977), 594–606.

[18] D. Mount, *CMSC 754: Computational geometry*, Spring 2000, Course Syllabus.

[19] J. O'Rourke, *Art gallery theorems and algorithms*, The International Series of Monographs on Computer Science, Oxford University Press, New York, 1987, QA830 O74a.

[20] ———, *Computational geometry in C*, Cambridge University Press, Cambridge, 1993, Second Edition, 1998.

[21] D.A. Plaisted e J. Hong, A heuristic triangulation algorithm, *Journal of Algorithms* **8** (1987), 405–437.

[22] F.P. Preparata e M.I. Shamos, *Computational geometry: An introduction*, Texts and Monographs in Computer Science, Springer-Verlag, New York, 1985, QA758 P927c.

[23] R. Seidel, A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons, *Computational Geometry: Theory and Aplications* **1** (1991), 51–64.

[24] M.I. Shamos e D. Hoey, Geometric intersection problems, *Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science*, 1976, pp. 208–215.

[25] R.E. Tarjan e C.J. Van Wyk, An $O(n\log\log n)$-time algorithm for triangulating a simple polygon, *SIAM Journal on Computing* **17** (1988), 143–178, Erratum in 17 (1988), 106.