1 Algoritmos Geométricos em SIG

Diversas funções de SIG dependem fundamentalmente de resultados obtidos em algumas disciplinas da computação, da matemática e da estatística, entre outras áreas. Especificamente na computação, as técnicas derivadas da área de bancos de dados são especialmente importantes, uma vez que são responsáveis pelos mecanismos de armazenamento e recuperação de dados geográficos. No entanto, como o diferencial do SIG está no uso de informação georreferenciada, em geral visualizável graficamente, duas outras disciplinas da computação adquirem grande importância nesse contexto: processamento digital de imagens e computação gráfica. A primeira é essencial para o uso de imagens em SIG, em aplicações que vão desde a conversão de dados até o sensoriamento remoto. A segunda reúne as técnicas de tratamento e visualização de dados vetoriais, que por sua vez se beneficiam dos avanços obtidos em uma área de pesquisas nova, porém importante: a *geometria computacional*.

A geometria computacional procura desenvolver e analisar algoritmos e estruturas de dados para resolver problemas geométricos diversos. Neste particular, tem um ponto importante de contato com a área de projeto e análise de algoritmos, uma vez que também procura caracterizar a dificuldade de problemas específicos, determinando a eficiência computacional dos algoritmos e usando técnicas de análise de complexidade assintótica [Knut73a]. Existe também uma preocupação em desenvolver soluções para problemas clássicos de geometria, construindo estruturas mais apropriadas para a representação geométrica robusta no ambiente computacional, que tem limitações conhecidas quanto à precisão numérica e a capacidade de armazenamento de dados.

Os mesmos problemas ocorrem na área de topologia, em que se procura desenvolver soluções, geralmente baseadas em estruturas de dados, para representar as relações espaciais que são independentes da geometria, tais como contenção, adjacência e conectividade. Além da caracterização e comparação das estruturas topológicas propriamente ditas, a geometria computacional preocupa-se com os algoritmos necessários para compor e manter estas estruturas a partir da geometria básica dos objetos.

Existe também muito desenvolvimento na área de indexação espacial para SIG. São algoritmos e estruturas de dados desenvolvidos especialmente para agilizar a busca e recuperação de dados em bancos de dados geográficos. As estruturas de indexação espacial procuram agilizar a resposta a dois tipos fundamentais de perguntas:

- dada uma região do espaço, identificar o que está contido nela;

- dado um ponto, determinar que objetos geográficos o contêm.

Com isso, a indexação espacial é utilizada a todo momento na operação de um SIG, em situações corriqueiras como a execução de um *zoom* ou a identificação de um objeto na tela com o *mouse*.

As seções seguintes abordam alguns dos principais algoritmos e estruturas de dados aplicados freqüentemente em SIG, nas áreas de geometria vetorial, topologia e indexação espacial.

1.1 Geometria computacional aplicada a SIG

Num sentido amplo, a geometria computacional compreende o estudo de algoritmos para resolver problemas geométricos em um computador. Nesta seção, os algoritmos geométricos utilizados para resolver problemas típicos de um SIG vetorial serão enfatizados, procurando transmitir uma noção dos recursos utilizados pelos sistemas na solução de problemas geográficos. O enfoque será o de mostrar *como* os algoritmos funcionam, indicando também, porém com menor ênfase, a sua complexidade.

1.1.1 Definições básicas

Em um SIG vetorial, cada objeto é codificado usando um ou mais pares de coordenadas, o que permite determinar sua localização e aparência visual. Adicionalmente, os objetos são também caracterizados por atributos não-espaciais, que os descrevem e identificam univocamente. Este tipo de representação não é exclusivo do SIG: sistemas CAD e outros tipos de sistemas gráficos também utilizam representações vetoriais. Isto porque o modelo vetorial é bastante intuitivo para engenheiros e projetistas, embora estes nem sempre utilizem sistemas de coordenadas ajustados à superfície da Terra para realizar seus projetos, pois para estas aplicações um simples sistema de coordenadas cartesianas é suficiente. Mas o uso de vetores em SIG é bem mais sofisticado do que o uso em CAD, pois em geral SIG envolve volumes de dados bem maiores, e conta com recursos para tratamento de topologia, associação de atributos alfanuméricos e indexação espacial. Por outro lado, os vetores que se constrói tipicamente em um SIG são menos sofisticados geometricamente que aqueles possíveis em um CAD. Enquanto em um SIG, em geral, se pode apenas representar pontos e conjuntos de segmentos de reta, em um CAD é possível ter também círculos, arcos de círculo, e curvas suavizadas como *spline* e Bezier. Além disto, o tratamento da terceira dimensão em SIG é ainda rudimentar, enquanto os sistemas CAD são utilizados para operações tridimensionais bem mais complexas, como modelagem de sólidos.

Para entender melhor a maneira como os SIG tratam a informação vetorial, estão relacionadas a seguir algumas definições fundamentais [*ref. Vetores em GIS*]. Como na maioria dos SIG comerciais, as definições consideram apenas duas dimensões.

Ponto: um *ponto* é um par ordenado *(x, y)* de coordenadas espaciais.

Alguns SIG denominam objetos localizados com apenas um ponto como *símbolos*. Isto se deve ao fato de sempre se associar um símbolo cartográfico ao ponto, para fins de apresentação em tela ou em um mapa.

Reta e segmento de reta: Sejam p_1 e p_2 dois pontos distintos no plano. A combinação linear $\alpha.p_1 + (1-\alpha)p_2$, onde α é qualquer número real, é uma *reta* no plano. Quando $0 \leq \alpha \leq 1$, se tem um *segmento de reta* no plano, que tem p_1 e p_2 como *pontos extremos*.

Esta definição é estritamente geométrica, e nos interessa uma definição mais aplicada. Assim, partimos para o conceito de *linha poligonal*, que é composta por uma seqüência de segmentos de reta. O mais comum, no entanto, é definir a linha poligonal através da seqüência dos pontos extremos de seus segmentos, ou seja, seus *vértices*.

Linha poligonal: Sejam $v_0, v_1, \ldots, v_{n-1}$ n pontos no plano. Sejam $s_0 = \overline{v_0 v_1}, s_1 = \overline{v_1 v_2}, \ldots, s_{n-2} = \overline{v_{n-2} v_{n-1}}$ uma seqüência de n - 1 segmentos, conectando estes pontos. Estes segmentos formam uma *poligonal* L se, e somente se, (1) a interseção de segmentos consecutivos é apenas o ponto extremo compartilhado por eles (i.e., $s_i \cap s_{i+1} = v_{i+1}$), (2) segmentos não consecutivos não se interceptam (i.e., $s_i \cap s_j = \varnothing$ para todo i, j tais que $j \neq i+1$), e (3) $v_0 \neq v_{n-1}$, ou seja, a poligonal não é fechada.

Observe-se, na definição acima, a exclusão da possibilidade de auto-interseção. Os segmentos que compõem a poligonal só se tocam nos vértices. Formalmente, poligonais que não obedecem a este critério são chamadas poligonais *complexas*. De modo geral, os SIG não impedem que poligonais complexas sejam criadas; no entanto, dificilmente este tipo de linha ocorrerá na natureza. Além do mais, poligonais complexas podem criar dificuldades na definição da topologia e em operações como a criação de *buffers* *(ref. Interna)*.

Polígono: Um *polígono* é a região do plano limitada por uma linha poligonal fechada.

A definição acima implica que, apenas invertendo a condição (3) da linha poligonal, temos um polígono. Assim, também aqui não é permitida a interseção de segmentos fora dos vértices, e os polígonos onde isto ocorre são denominados polígonos complexos. Os mesmos comentários que foram feitos para poligonais valem para os polígonos. Observe-se também que o polígono divide o plano em duas regiões: o interior, que convencionalmente inclui a fronteira (a poligonal fechada) e o exterior.

Assim, quando utilizamos a expressão *vetores*, estamos nos referindo a alguma combinação de pontos, poligonais e polígonos, conforme definidos acima. Combinações porque teoricamente poderíamos utilizar mais de um tipo de primitiva gráfica na criação da representação de um objeto. Por exemplo, pode-se ter objetos de área mais complexos, formados por um polígono básico e vários outros polígonos contidos no primeiro, delimitando buracos. Pode-se também ter objetos compostos por mais de um polígono, como seria necessário no caso do estado do Pará, que além da parte "continental" tem a ilha de Marajó e outras como parte de seu território.

1.1.1.1 Classes de vetores

Apesar de estarmos sempre concebendo representações sob a forma de pontos, linhas e áreas para objetos em SIG, existem algumas variações com relação à adaptação destas representações à realidade, ou seja, considerando a forma com que estes objetos

ocorrem na natureza. A opção entre as alternativas a seguir é feita na fase de modelagem conceitual do SIG, e deve ser feita com bastante cuidado.

Objetos de área podem ter três formas diferentes de utilização: como objetos *isolados*, objetos *aninhados* ou objetos *adjacentes*. O caso de objetos isolados é bastante comum em SIG urbanos, e ocorre no caso em que os objetos da mesma classe em geral não se tocam. Por exemplo, edificações, piscinas, e mesmo as quadras das aplicações cadastrais ocorrem isoladamente, não existindo segmentos poligonais compartilhados entre os objetos. O caso típico de objetos aninhados é o de curvas de nível e todo tipo de isolinhas, em que se tem linhas que não se cruzam, e são entendidas como estando "empilhadas" umas sobre as outras. Este caso tem muitas variações, pois curvas de nível podem ser também representadas como linhas, uma vez que podem permanecer abertas em algumas situações, e também podem ser entendidas como subproduto de modelos digitais de terreno, que são campos. Finalmente, temos objetos adjacentes, e os exemplos típicos são todas as modalidades de divisão territorial: bairros, setores censitários, municípios e outros. São também exemplos mapas geológicos e pedológicos, que representam fenômenos que cobrem toda a área de interesse. Neste caso, pode-se ter o compartilhamento de fronteiras entre objetos adjacentes, gerando a necessidade por estruturas topológicas. Estes também são os casos em que recursos de representação de buracos e ilhas são mais necessários.

Também objetos de linha podem ter variadas formas de utilização. Analogamente aos objetos de área, pode-se ter objetos de linha isolados, em árvore e em rede. Objetos de linha isolados ocorrem, por exemplo, na representação de muros e cercas em mapas urbanos. Objetos de linha organizados em uma árvore podem ser encontrados nas representações de rios e seus afluentes, e também em redes de esgotos e drenagem pluvial. E podem ser organizados em rede, nos casos de redes elétricas, telefônicas, de água ou mesmo na malha viária urbana e nas malhas rodoviária e ferroviária.

1.1.1.2 Problemas de nomenclatura

Um problema que aflige a todos os usuários de SIG é a grande variedade de diferentes nomenclaturas para elementos vetoriais. A linha poligonal, conforme definida, pode ser denominada de diversas formas em SIG e CAD: *linha*, *polilinha*, *arco*, *link*, *1-cell*, *cadeia*, e outras. Algumas destas denominações incluem considerações topológicas. Por exemplo, um arco é muitas vezes definido como um elemento que conecta dois nós e que pode ter ou não uma direção, e *nó* (ou *0-cell*) é uma denominação alternativa para *ponto* ou *símbolo*. O mesmo ocorre com relação a polígonos, denominados às vezes como *áreas*, *regiões* ou ainda *2-cells*.

Quase sempre aparecem sutilezas com relação à definição que serão especificamente ligadas a aspectos da lógica de construção do software SIG. Um SIG baseado em topologia, por exemplo, define áreas ou regiões a partir de seqüências de arcos, que por sua vez conectam nós. Um sistema *desktop mapping* poderá impedir a utilização de objetos compostos por vários polígonos. Um SIG baseado em SGBD relacional poderá permitir buracos, mas não permitir polígonos externos adicionais.

1.1.1.3 Tipos abstratos para dados vetoriais

Será apresentada a seguir um breve resumo das definições da seção anterior, seguida da a formulação de tipos abstratos de dados para suportar os dados vetoriais. Esta formulação será usada na descrição de algoritmos geométricos no restante deste capítulo.

Ponto: um ponto é um par ordenado *(x, y)* de coordenadas espaciais.

Linha poligonal: Sejam $v_0, v_1, \ldots, v_{n-1}$ n pontos no plano. Sejam $s_0 = \overline{v_0 v_1}, s_1 = \overline{v_1 v_2}, \ldots, s_{n-2} = \overline{v_{n-2} v_{n-1}}$ uma seqüência de n - 1 segmentos, conectando estes pontos. Estes segmentos formam uma *poligonal L* se, e somente se, (1) a interseção de segmentos consecutivos é apenas o ponto extremo compartilhado por eles (i.e., $s_i \cap s_{i+1} = v_{i+1}$), (2) segmentos não consecutivos não se interceptam (i.e., $s_i \cap s_j = \varnothing$ para todo i, j tais que $j \neq i+1$), e (3) $v_0 \neq v_{n-1}$, ou seja, a poligonal não é fechada.

Polígono: Um *polígono* é a região do plano limitada por uma linha poligonal fechada.

Estas três entidades geométricas básicas podem ser definidas em uma linguagem de programação usando tipos abstratos de dados, conforme apresentado no Programa 1.1. Essa definição inclui tipos abstratos para retângulos e para segmentos, que serão bastante úteis na indexação espacial e em alguns algoritmos geométricos. Não foi definido um tipo abstrato específico para polígonos, uma vez que corresponde a poligonais em que o primeiro e o último vértices coincidem. Para as poligonais, foi incluído no tipo uma variável `Retângulo`, para armazenar os limites do objeto em cada eixo[1].

[1] Este retângulo é usualmente denominado *retângulo envolvente mínimo* (REM), e é o menor retângulo com lados paralelos aos eixos que contém o objeto em questão.

```
estrutura Ponto
início
      inteiro x;
      inteiro y;
fim;

estrutura Segmento
início
      Ponto p1;
      Ponto p2;
fim;

estrutura Retângulo
início
      inteiro x1;
      inteiro y1;
      inteiro x2;
      inteiro y2;
fim;

estrutura Poligonal
início
      inteiro numPontos;
      Retângulo retânguloEnvolventeMínimo;
      Ponto[] vertice;
fim;
```

Programa 1.1 - Tipos abstratos de dados para `Ponto`, `Retângulo` e `Poligonal`

Um grande problema para a implementação de rotinas geométricas está relacionado com a precisão numérica. Como se sabe, a representação de números no computador é finita, uma vez que uma seqüência finita de bits apenas consegue representar uma seleção limitada de números de ponto flutuante [Schn97]. Esta limitação em geral não é considerada nos desenvolvimentos teóricos. O fechamento desta lacuna de precisão é deixado a cargo do programador, o que conduz a freqüentes problemas numéricos e de topologia nas aplicações reais.

Assim, em muitas situações, para minimizar o problema de precisão e melhorar o desempenho no tratamento da geometria, SIG e outros tipos de sistemas optam por representar coordenadas por meio de variáveis inteiras[2]. Isso viabiliza cálculos mais robustos e precisos, mas em contrapartida aparece a possibilidade de *overflow* numérico em determinadas circunstâncias. Em SIG, este problema torna-se ainda mais complicado devido aos sistemas de coordenadas utilizados mais freqüentemente. O sistema UTM (Universal Transverso de Mercator), por exemplo, divide o globo longitudinalmente em 60 *fusos*. Cada fuso cobre 6 graus de longitude, e é identificado por seu meridiano central. Em cada fuso, um sistema cartesiano de coordenadas é estabelecido, usando metros como unidades. O eixo *y* (ou seja, Norte) tem origem no Equador, e o eixo *x* (Leste) é posicionado de modo que a coordenada *x* seja equivalente a 500.000 metros sobre o meridiano central. As coordenadas *y*, que no hemisfério sul seriam negativas, são somadas a um fator constante de 10.000.000 metros. Assim, coordenadas UTM podem basicamente variar entre 0 e 1.000.000 metros em *x*, e entre 0 e 10.000.000

[2] Esta opção também é feita neste trabalho, considerando principalmente que é a alternativa mais usada pelos SIG comerciais.

metros em *y*. Para limitar a possibilidade de distorção na projeção, no entanto, os valores *x* variam de fato apenas de 166.667 a 666.667 metros.

Muitos SIG adotam inteiros de 32 bits, sem sinal, para representar coordenadas. Isto permite valores entre 0 e 4.294.967.295. Valores decimais são geralmente representados pela adoção de um número fixo de casas decimais, validos para todos os dados gráficos. Por exemplo, para representar coordenadas UTM com precisão de 1 centímetro, pode ser estabelecido um "fator de precisão" de 100. As coordenadas inteiras não mais representam metros, mas centímetros. O SIG assume a responsabilidade de inserir o ponto decimal na posição correta quando necessário, mas todos os cálculos internos serão realizados usando as coordenadas inteiras. Com um fator de 100, adequado para aplicações urbanas, a faixa de coordenadas efetivamente utilizáveis passa a ser de 0 a 42.949.672,95, que corresponde a mais de 40.000 quilômetros – suficiente para dar a volta ao mundo, e claramente mais do que o necessário para representar um fuso UTM.

No entanto, este limite nos força a prestar atenção a operações de multiplicação envolvendo coordenadas, ou valores derivados. Dependendo da ordem de grandeza dos valores *a* e *b*, o numerador da Equação 1.8 pode exceder os limites de representação. O valor de *a* pode se tornar arbitrariamente grande, à medida em que a reta se torna mais vertical. O valor de *b* também pode ser grande, dependendo do valor de *a* e da ordem de grandeza das coordenadas de um dos pontos extremos. Este problemas poderiam ser resolvidos com uma mudança de eixos, mas o processo seria muito complicado, computacionalmente intensivo e sujeito a erros. Seria também possível usar variáveis de ponto flutuante para estas operações, mas neste caso estariam sendo introduzidos erros de arredondamento e mais esforço computacional.

Um enfoque alternativo é apresentado em [Schn97], onde são descritas as bases da *geometria computacional de precisão* (ou *resolução*) *finita*. A idéia básica consiste em apoiar as coordenadas de quaisquer pontos ou vértices em uma malha regular baseada em valores inteiros. Qualquer ponto com coordenadas fracionárias (por exemplo, um ponto de interseção entre segmentos) é deslocado para o nó mais próximo desta grade regular. Este enfoque robustece bastante o tratamento numérico das coordenadas de pontos e vértices, e ainda garante consistência na representação topológica. No entanto, ainda não está presente em SIG comerciais.

1.1.2 Formulações e algoritmos básicos

1.1.2.1 Triângulos e produtos vetoriais

Diversos problemas de geometria computacional utilizam resultados básicos de problemas mais simples em sua solução. Alguns destes resultados básicos vêm da análise geométrica do mais simples dos polígonos, e o único que sempre é plano: o triângulo.

Área. Uma vez que na representação vetorial se trabalha com vértices e suas coordenadas, a fórmula elementar da geometria para cálculo da área de um triângulo ("a área de um triângulo é igual à metade do produto entre sua base e sua altura") não é muito prática. Em vez dela, serão utilizados dois resultados equivalentes da álgebra linear. O primeiro usa o produto de dois vetores, que determina a área de um

paralelogramo, o dobro da área do triângulo que interessa. Outro método calcula a área diretamente, por meio de um determinante 3x3.

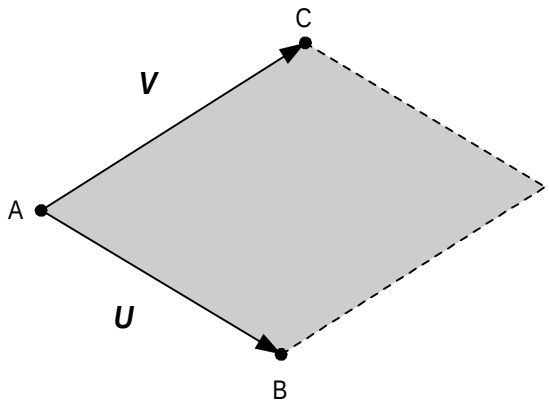


Figura 1.1 - Produto vetorial dos vetores *U* e *V*, equivalente ao dobro da área do triângulo *ABC*

O primeiro método pode ser descrito como se segue. Sejam *U* e *V* vetores. A área do paralelogramo com lados *U* e *V* é $|U \times V|$ (Figura 1.1). O produto vetorial pode ser calculado a partir do seguinte determinante:

$$\begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ x_U & y_U & z_U \\ x_V & y_V & z_V \end{vmatrix} = (y_U z_V - z_U y_V)\hat{i} + (z_U x_V - x_U z_V)\hat{j} + (x_U y_V - y_U x_V)\hat{k}$$

onde $\hat{i}, \hat{j}, \hat{k}$ são vetores unitários nas direções *x*, *y* e *z* respectivamente. Como se está tratando de vetores bidimensionais, temos $z_U = z_V = 0$, e portanto a área *S* do triângulo é dada por

$$S = \frac{(x_U y_V - y_U x_V)}{2}$$

Mas, na realidade, $U = B - A$, e $V = C - A$. Portanto, a expressão acima pode ser reescrita como

$$S = \frac{1}{2}(x_A y_B - y_A x_B + y_A x_C - x_A y_C + x_B y_C - y_B x_C) \qquad (1.1)$$

A área calculada pela expressão acima será positiva se os vértices *A*, *B* e *C* formarem um circuito em sentido anti-horário, e negativa se formarem um circuito no sentido horário. A área será exatamente zero se os três vértices estiverem alinhados.

A expressão acima pode ser também obtida quando se calcula o determinante dos três pares de coordenadas, substituindo a coordenada *z* por 1:

$$S = \frac{1}{2}\begin{vmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{vmatrix} = \frac{1}{2}(x_A y_B - y_A x_B + y_A x_C - x_A y_C + x_B y_C - y_B x_C) \qquad (1.2)$$

Também neste caso a área será negativa se a seqüência de vértices estiver orientada em sentido horário, e positiva caso contrário.

O cálculo efetivo da área de um triângulo, em números reais, desprezando o sinal, pode ser feito usando a função `áreaTriângulo` (Programa 1.2). Como pode ser interessante obter a área orientada, ou seja, com sinal, o Programa 1.2 também inclui a função `áreaOrientadaTriângulo.`

```
função áreaOrientadaTriângulo(Ponto A, Ponto B, Ponto C): real

início
      retorne ((A.x*C.y - A.y*C.x + A.y*B.x - A.x*B.y +
                C.x*B.y - C.y*B.x) / 2);
fim;

função áreaTriângulo(Ponto A, Ponto B, Ponto C): real

início
      retorne abs(áreaOrientadaTriângulo(A, B, C));
fim.
```

Programa 1.2 - Funções `áreaTriângulo` e `áreaOrientadaTriângulo`

Coordenadas baricêntricas. Para determinar se um determinado ponto pertence ou não a um triângulo, utiliza-se um método baseado em *coordenadas baricêntricas* [FiCa91].

> ***Teorema 1.1*** *- Sejam p_1, p_2 e p_3 pontos não colineares no plano. Então cada ponto p do plano pode ser escrito na forma*
>
> $$p = \lambda_1 p_1 + \lambda_2 p_2 + \lambda_3 p_3 \qquad (1.3)$$
>
> *onde λ_1, λ_2 e λ_3 são números reais e $\lambda_1 + \lambda_2 + \lambda_3 = 1$. Os coeficientes λ_1, λ_2 e λ_3 são denominados* coordenadas baricêntricas *de p em relação a p_1, p_2 e p_3.*
>
> ***Prova*** **-** Com as coordenadas dos pontos *p*, p_1, p_2 e p_3, e a equação $\lambda_1 + \lambda_2 + \lambda_3 = 1$, constrói-se um sistema de três equações e três incógnitas para encontrar as coordenadas baricêntricas:
>
> $$\begin{aligned} \lambda_1 x_1 &+ \lambda_2 x_2 &+ \lambda_3 x_3 &= x_p \\ \lambda_1 y_1 &+ \lambda_2 y_2 &+ \lambda_3 y_3 &= y_p \end{aligned}$$
>
> O sistema acima tem por determinante exatamente aquele apresentado na Equação 1.2 e seu valor corresponde ao dobro da área do triângulo $p_1p_2p_3$. A área é não-nula, pois p_1, p_2 e p_3 não são alinhados por hipótese. Assim, o sistema tem solução única para cada *p*.

Os valores de λ_1, λ_2 e λ_3 podem ser obtidos usando a regra de Cramer, e expressos em termos de áreas de triângulos. Temos, portanto:

$$\lambda_1 = \frac{S(pp_2p_3)}{S(p_1p_2p_3)},\ \lambda_2 = \frac{S(p_1pp_3)}{S(p_1p_2p_3)} \text{ e } \lambda_3 = \frac{S(p_1p_2p)}{S(p_1p_2p_3)}$$

A análise do sinal das coordenadas baricêntricas indica a região do plano em que se encontra *p*, em relação ao triângulo $p_1p_2p_3$ (Figura 1.2). Observe-se que, para isso, as áreas devem ser orientadas, ou seja, com sinal.

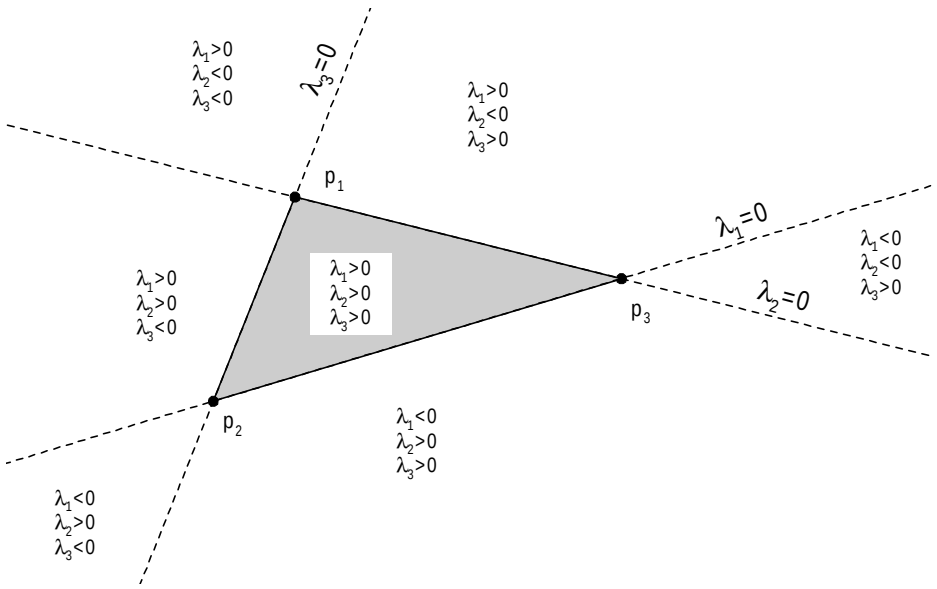


Figura 1.2 - Sinais das coordenadas baricêntricas

Este resultado leva à implementação de uma função bastante útil, que determina se um ponto está contido em um triângulo (Programa 1.3).

```
função pontoEmTriângulo(Ponto P,
                        Ponto P1, Ponto P2, Ponto P3):  booleano

início
      real lambda1, lambda2, lambda3, S;

      S = áreaOrientadaTriângulo(P1, P2, P3);
      lambda1 = áreaOrientadaTriângulo(P, P2, P3) / S;
      lambda2 = áreaOrientadaTriângulo(P1, P, P3) / S;
      lambda3 = áreaOrientadaTriângulo(P1, P2, P) / S;

      retorne ((lambda1 > 0) e (lambda2 > 0) e (lambda3 > 0))
fim.
```

Programa 1.3 - Função `pontoEmTriângulo`

1.1.2.2 Pontos e segmentos

Schneider [Schn97] define exaustivamente os tipos de situações de posicionamento relativo entre pontos e segmentos de reta, por meio de predicados. Estes predicados estão listados na Tabela 1.1 e na Tabela 1.2.

Tabela 1.1 - Posicionamento relativo de ponto e segmento

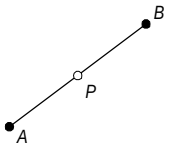
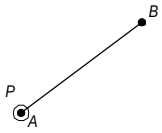
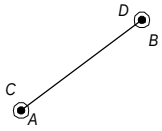
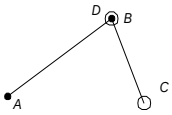
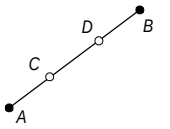
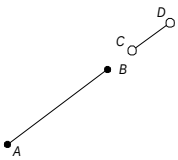
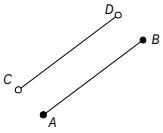
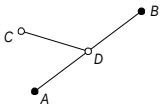
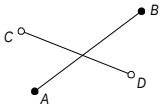
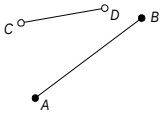
	`em(P, AB)`	Ponto é interior ao segmento; pontos extremos são excluídos.
	`emExtremo(P, AB)`	Ponto coincide com um ponto extremo do segmento

Tabela 1.2 - Posicionamento relativo entre dois segmentos

	`iguais(AB, CD)`	Ambos os pontos extremos coincidem
	`seEncontram(AB, CD)`	Compartilham exatamente um ponto extremo
	`superpostos(AB, CD)`	São colineares e compartilham um trecho comum
	`alinhados(AB, CD)`	São colineares e não têm ponto em comum
	`paralelos(AB, CD)`	Têm a mesma inclinação e não são iguais nem superpostos
	`seTocam(AB, CD)`	Não são superpostos e um dos pontos extremos de um segmento pertence ao outro segmento
	`seInterceptam(AB, CD)`	Têm um ponto em comum e não se encontram nem se tocam
	`disjuntos(AB, CD)`	Não são iguais, nem se encontram, nem se tocam, nem são paralelos, nem se interceptam, nem se superpõem

Adicionalmente aos predicados listados acima, existe a necessidade de definir uma única função, denominada `pontoInterseção`, que retornará as coordenadas do ponto de interseção (se houver) entre dois segmentos.

A implementação dos predicados depende de funções mais básicas. Uma delas, a função para detectar o posicionamento relativo entre ponto e segmento orientado de reta, é baseada no sinal do produto vetorial, conforme apresentado na seção 1.1.2. Por exemplo, para determinar se o ponto *C* está à direita ou à esquerda do segmento orientado *AB*, basta calcular a área do triângulo *ACB* pela Equação 1.1. Se esta for positiva, o ponto *C* está à esquerda (Figura 1.3a); se for negativa, *C* está à direita (Figura 1.3b). Se a área calculada for nula, então *A*, *B* e *C* estão alinhados (Figura 1.3c). Naturalmente, para esta e outras aplicações, é desnecessário calcular a área: apenas o sinal do produto vetorial interessa.

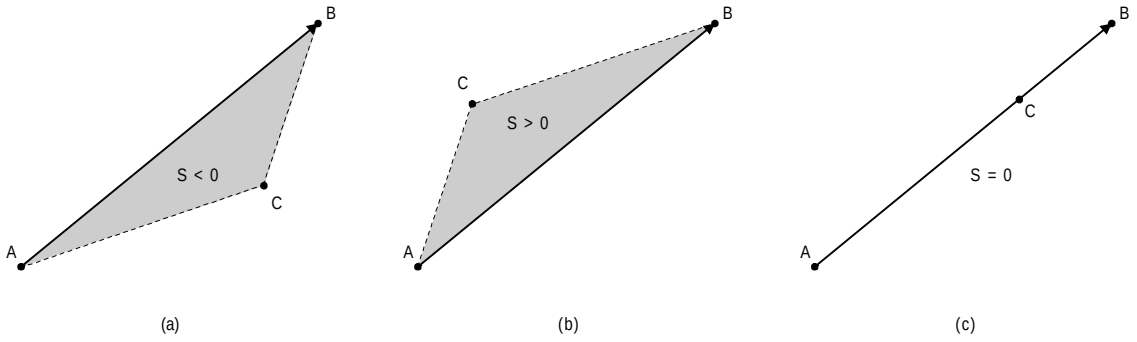


Figura 1.3 - Posicionamento relativo de ponto e segmento orientado

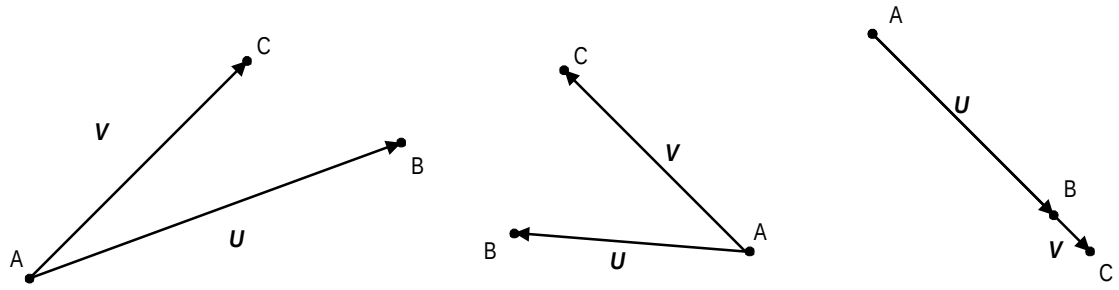
Uma implementação possível para este teste está apresentada no Programa 1.4. Observe-se que, em comparação com a função `áreaTriângulo`, o cálculo da área está incompleto: não há a necessidade de efetuar a divisão por 2.

```
função lado(Ponto A, Ponto B, Ponto C): inteiro
/* determina se C está à direita, à esquerda ou alinhado com AB */
/* direita: retorna -1; esquerda: retorna 1; alinhado: retorna 0 */
início
      inteiro S;

      S = A.x*C.y - A.y*C.x + A.y*B.x - A.x*B.y + C.x*B.y - C.y*B.x;
      se (S < 0) então retorne -1;
      se (S > 0) então retorne 1 senão retorne 0;
fim.
```

Programa 1.4 - Função `lado`

A mesma formulação vale para determinar a posição relativa entre dois vetores *U* e *V*. Se o resultado do produto vetorial $U \times V$ for positivo, então o giro de *U* a *V* é anti-horário (Figura 1.4a); caso contrário, o giro é no sentido horário (Figura 1.4b). Também aqui, se o resultado for nulo, significa que os vetores são colineares (Figura 1.4c). Este resultado é importante para a ordenação de raios no algoritmo de fecho convexo de Graham [Sedg90] (vide seção 1.1.6).



(a) $\boldsymbol{U} \times \boldsymbol{V} > 0$ (b) $\boldsymbol{U} \times \boldsymbol{V} < 0$ (c) $\boldsymbol{U} \times \boldsymbol{V} = 0$

Figura 1.4 - Posicionamento relativo entre dois vetores

Com a função `lado`, somada a um teste simples de coincidência de pontos, denominado `sobre` (Programa 1.5), pode-se implementar os dois predicados da Tabela 1.1 (Programa 1.6) e, utilizando estes, pode-se implementar alguns dos predicados relacionados na Tabela 1.2, como `iguais`, `seEncontram`, `superpostos`, `alinhados` e `seTocam` (Programa 1.7).

```
função sobre(Ponto A, Ponto B): booleano
/* testa se A e B coincidem */
início
      retorne ((P.x = A.x) e (P.y = A.y));
fim.
```

Programa 1.5 - Função `sobre`

```
função em(Ponto P, Ponto A, Ponto B): booleano
/* testa se P está "dentro" de AB, mas não coincide com A ou B */
início
      se (lado(A, B, P) = 0)  /* P pertence à reta AB */
      então início

            /* se AB não for vertical, testar em x; senao, em y */
            se (A.x != B.x)
            então retorne (((A.x < P.x) e (P.x < B.x)) ou
                           ((A.x > P.x) e (P.x > B.x)) )

            senão retorne (((A.y < P.y) e (P.y < B.y)) ou
                           ((A.y > P.y) e (P.y > B.y)) );

      fim então
      senão
            retorne falso;
fim.

função extremo(Ponto P, Ponto A, Ponto B): booleano
/* testa se P coincide com um ponto extremo de AB */
início
      retorne (sobre(P, A) ou sobre(P, B));
fim.
```

Programa 1.6 - Funções de comparação entre ponto e segmento

```
função iguais(Ponto A, Ponto B, Ponto C, Ponto D): booleano
/* testa se AB e CD são iguais (coincidentes) */
início
      retorne ((sobre(A, C) e sobre(B, D)) ou
               (sobre(A, D) e sobre(B, C)));
fim.

função seEncontram(Ponto A, Ponto B, Ponto C, Ponto D): booleano
/* testa se AB e CD se encontram (um ponto extremo coincidente) */
início
      se iguais(A, B, C, D) então retorne falso;

      retorne ((sobre(A, C) e não em(D, A, B) e não em(B, C, D)) ou
               (sobre(A, D) e não em(C, A, B) e não em(B, C, D)) ou
               (sobre(B, C) e não em(D, A, B) e não em(A, C, D)) ou
               (sobre(B, D) e não em(C, A, B) e não em(A, C, D));
fim.

função superpostos(Ponto A, Ponto B, Ponto C, Ponto D): booleano
/* testa se AB e CD são alinhados e têm um trecho em comum */
início
      se ((lado(A, B, C) = 0) e (lado(A, B, D) = 0))
      então retorne (em(C, A, B) ou em(D, A, B) ou
                     em(A, C, D) ou em(B, C, D));
fim.

função alinhados(Ponto A, Ponto B, Ponto C, Ponto D): booleano
/* testa se AB e CD são alinhados e não têm um trecho em comum */
início
      se ((lado(A, B, C) = 0) e (lado(A, B, D) = 0))
      então retorne (não em(C, A, B) e não em(D, A, B) e
                     não em(A, C, D) e não em(B, C, D));
fim.

função seTocam(Ponto A, Ponto B, Ponto C, Ponto D): booleano
/* testa se AB e CD se tocam */
início
      se (alinhados(A, B, C, D) ou superpostos(A, B, C, D))
      então retorne falso
      senão retorne (em(C, A, B) ou em(D, A, B) ou
                     em(A, C, D) ou em(B, C, D));
fim.
```

Programa 1.7 - Funções `iguais`, `seEncontram`, `superpostos`, `alinhados` e `seTocam`

No caso do predicado `paralelos`, a solução mais simples consiste em calcular e comparar os coeficientes angulares das retas que contêm os segmentos, tomando os cuidados necessários para o caso de retas verticais. Para evitar problemas numéricos, pode-se usar a formulação descrita abaixo.

Para que a reta *AB* seja paralela à reta *CD*, devemos ter

$$\frac{y_B - y_A}{x_B - x_A} = \frac{y_D - y_C}{x_D - x_C}$$

e portanto

$$(y_B - y_A)\cdot(x_D - x_C) - (y_D - y_C)\cdot(x_B - x_A) = 0 \qquad (1.4)$$

Desenvolvendo, temos uma forma mais elegante, e mais freqüente na literatura:

$$x_A(y_D - y_C) + x_B(y_C - y_D) + x_C(y_A - y_B) + x_D(y_B - y_A) = 0 \qquad (1.5)$$

Note-se na Equação 1.5 que apenas são realizadas operações de multiplicação e subtração, eliminando a necessidade de testes para casos especiais, como retas verticais. A implementação da função `paralelos` está no Programa 1.8. É dada preferência na implementação à Equação 1.4, uma vez que, quando as operações de subtração são realizadas antes da multiplicação, corre-se menos risco de *overflow*. Observe-se os testes de alinhamento, superposição e igualdade realizados no início da função, necessários para eliminar as outras três situações em que o coeficiente angular das retas também coincide.

```
função paralelos(Ponto A, Ponto B, Ponto C, Ponto D): booleano
/* testa se A e B são paralelos */
início
      se (alinhados(A, B, C, D) ou
          superpostos(A, B, C, D) ou
          iguais(A, B, C, D))
      então retorne falso;

      retorne (((B.y - A.y) * (D.x - C.x) -
                (D.y - C.y) * (B.x - A.x)) = 0);
fim.
```

Programa 1.8 - Função `paralelos`

Existem diversas maneiras de implementar o predicado `seInterceptam`. A alternativa mais conveniente consiste em usar um método baseado no produto vetorial para detectar a interseção entre dois segmentos, fazendo o exame dos triângulos que podem ser formados por quaisquer três dos quatro pontos extremos. Será utilizada para isso a função `lado`, descrita anteriormente (Programa 1.4).

Cuidados especiais deverão ser tomados para garantir que a interseção é *própria*. Se o ponto de interseção não é pertence a pelo menos um dos segmentos, diz-se que a interseção é *imprópria* (Figura 1.6a). Quando o ponto de interseção é interior a ambos os segmentos, como na Figura 1.5, a interseção é *própria*. O conceito de interseção própria em geral não inclui o caso em que a interseção ocorre em um dos pontos extremos (Figura 1.6b), que corresponde ao caso dos predicados `seTocam` e `seEncontram`.

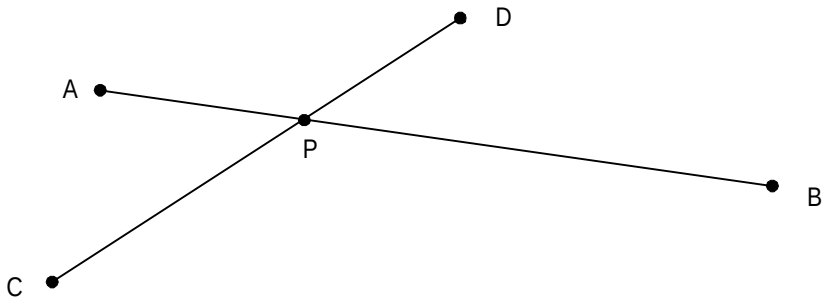


Figura 1.5 - Interseção própria de segmentos

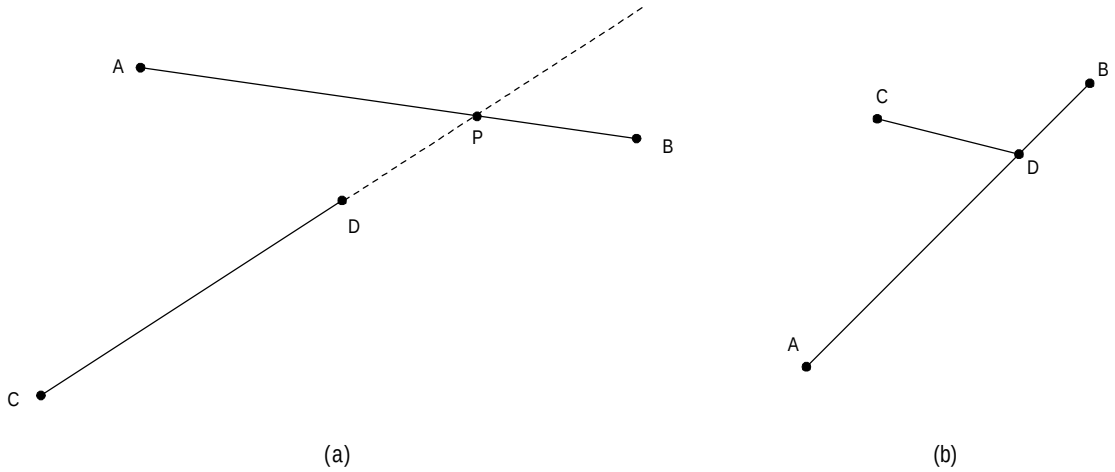


Figura 1.6 - Interseção imprópria de segmentos

A implementação do teste de interseção entre dois segmentos pode ser dividida em duas etapas [CLR90]. Inicialmente, é utilizado um teste rápido, para determinar se os retângulos definidos pelos segmentos se tocam. Se os retângulos não se tocarem em *x* ou em *y*, os segmentos também não terão interseção, mas não se pode afirmar o contrário (Figura 1.7b). Este teste é semelhante ao de interseção própria. Os segmentos *AB* e *CD poderão* se tocar (Figura 1.7a) caso

$$(x_2 \geq x_3) \wedge (x_4 \geq x_1) \wedge (y_2 \geq y_3) \wedge (y_4 \geq y_1)$$

onde

$$x_1 = min(x_A, x_B) \qquad x_3 = min(x_C, x_D)$$
$$x_2 = max(x_A, x_B) \qquad x_4 = max(x_C, x_D)$$
$$y_1 = min(y_A, y_B) \qquad y_3 = min(y_C, y_D)$$
$$y_2 = max(y_A, y_B) \qquad y_4 = max(y_C, y_D)$$

ou seja, $P = (x_1, y_1)$ e $Q = (x_2, y_2)$ são respectivamente o canto inferior esquerdo e superior direito do REM de *AB*, e $P' = (x_3, y_3)$ e $Q' = (x_4, y_4)$ são, analogamente, os cantos do REM de *CD* (Figura 1.7).

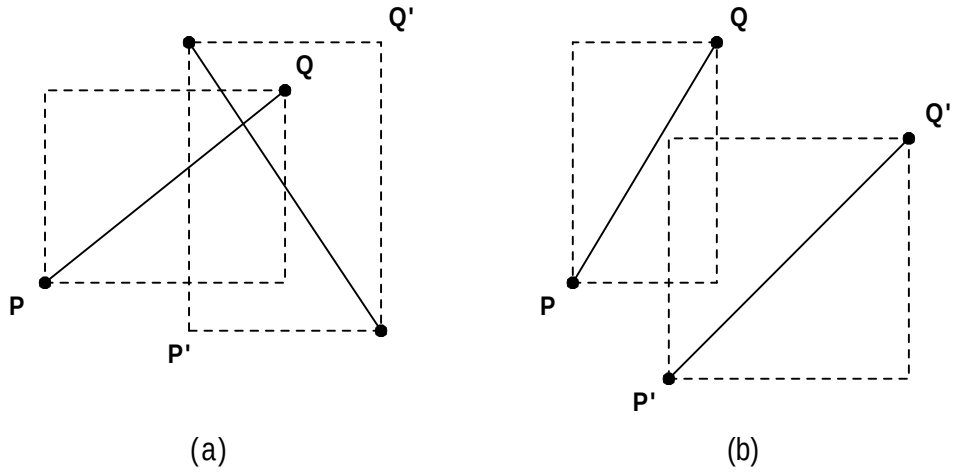


Figura 1.7 - Interseção de retângulos envolventes mínimos

A implementação deste teste, a função `interseçãoRetângulos`, está apresentada no Programa 1.9.

```
função interseçãoRetângulos(Ponto A, Ponto B, Ponto C, Ponto D):
   booleano

início
      Ponto P, Ponto Q, Ponto P1, Ponto Q1;

      P.x = min(A.x, B.x);
      P.y = min(A.y, B.y);
      Q.x = max(A.x, B.x);
      Q.y = max(A.y, B.y);
      P1.x = min(C.x, D.x);
      P1.y = min(C.y, D.y);
      Q1.x = max(C.x, D.x);
      Q1.y = max(C.y, D.y);

      retorne ((Q.x >= P1.x) e (Q1.x >= P.x) e
               (Q.y >= P1.y) e (Q1.y >= P.y));
fim.
```

Programa 1.9 - Interseção de retângulos envolventes mínimos

O segundo estágio consiste em verificar se os segmentos efetivamente se interceptam. Isto ocorre quando os pontos extremos de um segmento ficam de lados opostos da reta definida pelo outro, e vice-versa. Os resultados do produto vetorial têm que ter sinais opostos (Figura 1.8a). Se apenas um dos produtos for nulo, então um ponto extremo de um segmento está contido na reta definida pelo outro (Figura 1.8b). Se ambos os produtos forem nulos, os segmentos são colineares (Figura 1.8c), com interseção (a possibilidade de colinearidade sem interseção foi descartada pelo teste dos retângulos). O teste precisa ser aplicado duas vezes, usando cada segmento como base, ou seja, não basta verificar se *C* e *D* estão de lados opostos da reta definida por *AB*, também é preciso verificar se *A* e *B* estão de lados opostos da reta *CD*.

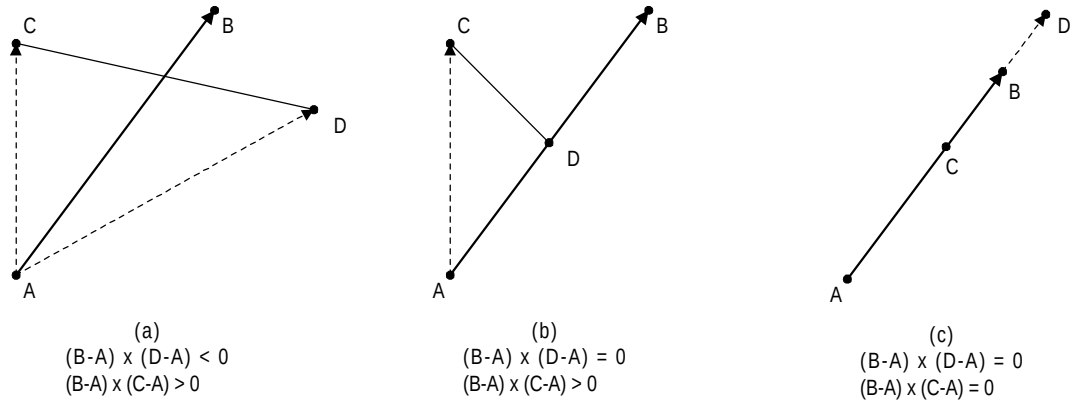


Figura 1.8 - Verificação de interseção

Para implementar este teste, utiliza-se o teste de posicionamento relativo entre ponto e reta (função `lado`), gerando o predicado `seInterceptam` (Programa 1.10). Nesta função está contida uma chamada ao teste rápido de interseção de retângulos, para que os produtos vetoriais só sejam calculados caso estritamente necessário.

```
função seInterceptam(Ponto A, Ponto B, Ponto C, Ponto D): booleano

início
      inteiro abc, abd, cda, cdb;

      se não interseçãoRetângulos(A, B, C, D)
      então retorne falso;

      abc = lado(A, B, C);
      abd = lado(A, B, D);
      cda = lado(C, D, A);
      cdb = lado(C, D, B);

      retorne ((abc * abd < 0) e (cda * cdb < 0));
fim.
```

Programa 1.10 - Função `seInterceptam`

Se, em alguma situação, o caso de interseção em um dos pontos extremos puder ser considerado, basta incluir uma chamada às funções `seEncontram` e `seTocam`, gerando uma nova função, `seInterceptamImprópria` (Programa 1.11).

```
função seInterceptamImprópria(Ponto A, Ponto B, Ponto C, Ponto D):
     booleano

início
      retorne (seEncontram(A, B, C, D) ou
               seTocam(A, B, C, D) ou
               seInterceptam(A, B, C, D));
fim.
```

Programa 1.11 - Função `seInterceptamImprópria`

O predicado restante, `disjuntos`, é implementado com testes para os demais casos, chegando a uma conclusão por exclusão: dois segmentos são disjuntos se não são iguais, nem se encontram, nem se tocam, nem são paralelos, nem se interceptam, nem se superpõem (Programa 1.12).

```
função disjuntos(Ponto A, Ponto B, Ponto C, Ponto D): booleano

início
      retorne (não iguais(A, B, C, D) e
               não seEncontram(A, B, C, D) e
               não seTocam(A, B, C, D) e
               não paralelos(A, B, C, D) e
               não superpostos(A, B, C, D) e
               não seInterceptam(A, B, C, D));
fim.
```

Programa 1.12 - Função `disjuntos`

A função `seInterceptam` serve para determinar, rapidamente e com o mínimo de problemas numéricos possível, *se* um par de segmentos tem interseção, o que é suficiente em diversas situações. O cálculo das coordenadas deste ponto de interseção exige um pouco mais de esforço.

A primeira e mais óbvia solução é baseada em geometria analítica. Dados dois pares de pontos, cada par definindo um segmento no plano, deve-se deduzir as equações das retas e resolver um sistema de duas equações e duas incógnitas, cuja solução é o ponto de interseção.

A formulação é a seguinte: sejam *AB* e *CD* dois segmentos de reta quaisquer no plano (Figura 1.5). A reta que passa por *A* e *B* tem a seguinte equação:

$$y = a_1 x + b_1 . \tag{1.6}$$

Como na reta *AB* temos

$$\frac{y - y_A}{x - x_A} = \frac{y_B - y_A}{x_B - x_A},$$

então

$$a_1 = \frac{y_B - y_A}{x_B - x_A} \text{ e } b_1 = y_A - a_1 x_A .$$

O mesmo se aplica à reta *CD*, produzindo

$$y = a_2 x + b_2 , \tag{1.7}$$

e

$$a_2 = \frac{y_D - y_C}{x_D - x_C} \text{ e } b_2 = y_C - a_2 x_C .$$

Resolvendo o sistema formado pelas equações 1.6 e 1.7, temos

$$x = \frac{b_2 - b_1}{a_1 - a_2} \text{ e } y = \frac{b_2 a_1 - b_1 a_2}{a_1 - a_2} \tag{1.8}$$

A solução para o sistema indica o ponto de interseção entre as *retas AB* e *CD*, que não necessariamente pertence aos *segmentos AB* e *CD*. Assim, um teste adicional é necessário para verificar se a interseção é ou não própria. Isto pode ser feito verificando se ambas as coordenadas do ponto de interseção *P* estão dentro do intervalo formado pelas coordenadas dos pontos extremos de ambos os segmentos, ou seja:

$$min(x_A, x_B) \leq x_P \leq max(x_A, x_B) \text{ e } min(y_A, y_B) \leq y_P \leq max(y_A, y_B)$$

e também

$$min(x_C, x_D) \leq x_P \leq max(x_C, x_D) \text{ e } min(y_C, y_D) \leq y_P \leq max(y_C, y_D) .$$

É necessário permitir a condição de igualdade, nos testes acima, de modo a considerar o caso de retas verticais ou horizontais. Portanto, para garantir que a interseção seja própria, será ainda necessário comparar diretamente o ponto de interseção com os pontos extremos dos segmentos.

Embora a formulação acima pareça ser uma boa maneira de resolver o problema, alguns obstáculos e inconveniências persistem. Em primeiro lugar, um programa que implemente esta formulação precisa lidar com a possibilidade de se obter zero em um dos denominadores. Isto pode acontecer em três situações: (1) a reta *AB* é vertical ($x_A = x_B$), (2) a reta *CD* é vertical ($x_C = x_D$), ou (3) as retas *AB* e *CD* são paralelas ($a_1 = a_2$). Nos casos (1) e (2), o ponto de interseção pode ser calculado de forma trivial, aplicando a abscissa da reta vertical à equação da outra reta. No caso (3) fica claro que as retas não têm ponto de interseção, e portanto o problema não tem solução, *i.e.*, não existe ponto de interseção, próprio ou impróprio. Em todos os casos, no entanto, mas problemas numéricos ainda podem surgir quando o denominador é "quase" zero. Lidar com estas exceções no código pode ser problemático, levando ao desenvolvimento de programas menos robustos.

Uma solução um pouco mais conveniente para o problema usa também a geometria analítica, mas aplica uma representação paramétrica para os segmentos que evitará as inconveniências listadas acima. Esta formulação é a seguinte: seja *AB* um segmento de reta qualquer no plano. Seja $U = B - A$ um vetor correspondente ao segmento. Qualquer ponto *P* ao longo da *reta* que passa por *A* e *B* pode ser obtido a partir da soma vetorial $P(s) = A + sU$, onde *s* é denominado o *parâmetro* da equação. Mas temos que $P(0) = A$, e $P(1) = A + U = B$, e é possível chegar a qualquer ponto do segmento *AB* variando *s* entre 0 e 1.

Analogamente, pode-se representar o segmento *CD* pela expressão $Q(t) = C + tV$, onde $V = D - C$ e *t* é o parâmetro. Um ponto de interseção entre *AB* e *CD* é aquele para o qual $P(s) = Q(t)$, e portanto $A + sU = C + tV$. Decompondo os pontos e os vetores em coordenadas, temos novamente um sistema de duas equações e duas incógnitas em *s* e *t*, cuja solução é [ORou94]:

$$\begin{aligned} s &= \frac{x_A(y_D - y_C) + x_C(y_A - y_D) + x_D(y_C - y_A)}{denom} \\ t &= -\frac{\left(x_A\left(y_C - y_B\right) + x_B\left(y_A - y_C\right) + x_C\left(y_B - y_A\right)\right)}{denom} \end{aligned} \tag{1.9}$$

onde

$$denom = x_A\left(y_D - y_C\right) + x_B\left(y_C - y_D\right) + x_C\left(y_A - y_B\right) + x_D\left(y_B - y_A\right)$$

As coordenadas do ponto de interseção *I* serão:

$$\begin{aligned} x_I &= x_A + s\left(x_B - x_A\right) \\ y_I &= y_A + s\left(y_B - y_A\right) \end{aligned} \tag{1.10}$$

mas a interseção somente será própria se $0 < s < 1$ e $0 < t < 1$.

Observe-se que o valor de *denom* é equivalente à expressão à esquerda da Equação 1.5, e mais uma vez é mais conveniente implementá-la sob a forma da Equação 1.4. Se *denom* for igual a zero, então os segmentos são paralelos e não existe ponto de interseção. No caso, é mais conveniente calcular o valor de *denom* e não utilizar a função `paralelas` para detectar este caso especial, uma vez que *denom* será utilizado no cálculo de *s* e *t*.

A implementação está no Programa 1.13. A função `pontoInterseção` retornará um valor booleano, indicando se existe ou não o ponto de interseção próprio, enquanto o ponto propriamente dito, se existir, será retornado na variável `I`.

Em uma tentativa de evitar que o cálculo seja feito inutilmente, ou seja, para evitar que um ponto de interseção impróprio seja calculado, inseriu-se no início da função um teste preliminar, usando o predicado `seInterceptam`. Note-se também que os problemas numéricos não são totalmente eliminados com esta implementação. Ainda existe o risco, por exemplo, de se ter um valor de *denom* exageradamente alto, correspondente à situação em que as retas são quase paralelas, podendo gerar imprecisão numérica no cálculo de *s* e *t*. Faz sentido, portanto, assumir o custo adicional do teste `seInterceptam`, que incorpora o teste simples e seguro de interseção de retângulos, para evitar ao máximo situações numericamente problemáticas.

```
função pontoInterseção(Ponto A, Ponto B, Ponto C, Ponto D, Ponto I):
     booleano

início
      real s, t, denom;

      se não seInterceptam(A, B, C, D)
      então retorne falso;

      denom = ((B.y - A.y) * (D.x - C.x) - (D.y - C.y) * (B.x - A.x));
      se (denom = 0)
      então retorne falso;

      s = (A.x * (D.y - C.y) +
           C.x * (A.y - D.y) +
           D.x * (C.y - A.y)) / denom;
      t = - (A.x * (C.y - B.y) +
             B.x * (A.y - C.y) +
             C.x * (B.y - A.y) / denom;

      se ((s > 0) e (s < 1) e (t > 0) e (t < 1))
      então início
            I.x = A.x + s * (B.x - A.x);
            I.y = A.y + s * (B.y - A.y);
            retorne verdadeiro;
      fim
      senão retorne falso;
fim.
```

Programa 1.13 - Função `pontoInterseção`

1.1.2.3 Robustez numérica das implementações

Como já mencionado, um dos principais problemas que afetam os algoritmos geométricos é a robustez numérica da implementação. Como se sabe, a representação de números reais em computador é limitada, em termos de precisão. No entanto, o desenvolvimento teórico dos algoritmos é baseada na hipótese de que se dispõe de operadores aritméticos com precisão infinita, ignorando em sua concepção os possíveis erros de arredondamento.

No momento da implementação prática, o programador em geral ignora este problema, e passa a conviver com uma série de problemas que são tratados caso a caso. Isso ajuda a explicar parte da instabilidade de comportamento que infesta os SIG comerciais, especialmente quando se trata de operações sobre objetos geográficos vetoriais. A correção destes problemas é feita, em geral, introduzindo "tolerâncias" em lugar de testes exatos. Por exemplo, a comparação "`se denom = 0`" na função `pontoInterseção` pode ser implementada na prática como "`se denom` $< \varepsilon$", para evitar *overflow* numérico na expressão seguinte, e para diminuir prováveis problemas de arredondamento. Ainda assim, existe dúvida sobre qual valor de ε utilizar.

A concepção e a descrição dos algoritmos apresentados na seção anterior procurou evitar ao máximo possíveis problemas numéricos. Dentre todas as funções apresentadas na seção anterior, apenas três contém riscos relativos à robustez numérica de implementação: as funções `lado`, `paralelos` e `pontoInterseção` Todas as outras funções que poderiam apresentar problemas numéricos, como `em`, `superpostos` e `alinhados`, utilizam diretamente uma dessas funções. O problema

numérico da função `lado` está na exatidão do teste de alinhamento dos três pontos, análogo ao cálculo da diferença entre inclinações na função `paralelos` e ao cálculo de *denom* em `pontoInterseção`. Seria possível estabelecer uma tolerância para as comparações nestas situações? No caso de aplicações de SIG, a tolerância precisaria variar de acordo com o sistema de coordenadas e com a escala da representação. Uma possibilidade também seria a determinação da tolerância por parte do usuário, considerando estes fatores, para cada camada de informação vetorial.

O'Rourke [ORou94] observa que a introdução generalizada de tolerâncias pode causar problemas em outras áreas, como na detecção de pontos coincidentes (função `sobre`), e sugere que o único método infalível de solução seria implementar os algoritmos usando números racionais, ao preço de tornar o código excessivamente complexo. Novamente, o caminho de solução mais interessante parece ser o proposto pela geometria computacional de precisão finita [Schn97], mas ao custo de uma maior complexidade na implementação de funções básicas como as de interseção entre dois segmentos.

1.1.3 Interseção de *n* segmentos

Informalmente, este problema pode ser enunciado da seguinte maneira: dados *n* segmentos de reta no plano, determinar se existe alguma interseção entre quaisquer dois destes segmentos. Um problema correlato seria: dados *n* segmentos de reta no plano, determinar todas as interseções que ocorram.

A idéia para solução do primeiro problema vem da análise de intervalos em uma dimensão. Considere-se que, em vez de *n* segmentos, tenha-se *n* intervalos entre números reais, do tipo $[x_L, x_R]$, onde $x_L \leq x_R$. Uma solução exaustiva seria analisar todos os n^2 pares de intervalos existentes, comparando-os sempre dois a dois, e interrompendo o processamento assim que a primeira interseção fosse detectada.

No entanto, uma maneira mais eficiente de resolver o problema é construir uma lista ordenada dos valores extremos dos intervalos, tomando o cuidado de identificá-los como sendo *L* ou *R*, de acordo com sua situação no intervalo. Assim, não haverá interseção alguma entre os intervalos se e somente se a lista ordenada contiver uma seqüência alternada de *Ls* e *Rs*: *L R L R ... L R L R*. Em qualquer outra situação, pode-se afirmar que existe superposição entre algum par de intervalos (Figura 1.9). Esta solução tem complexidade computacional da ordem de $O(n \log n)$, uma vez que é dominada pela ordenação dos valores extremos [PrSh88].

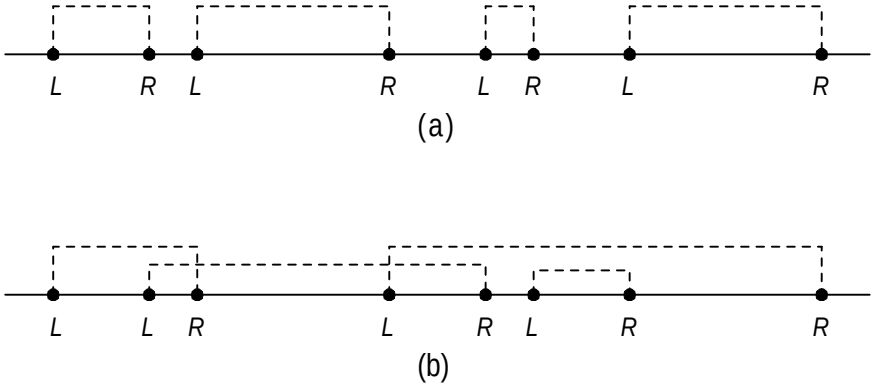


Figura 1.9 - Verificação de interseção em intervalos na reta

Em duas dimensões, o problema torna-se um pouco mais complicado, já que não existe maneira de produzir uma ordenação adequada para segmentos no plano. A técnica empregada é clássica na geometria computacional, e é denominada de *varredura do plano* (*plane sweep*). Esta técnica faz uso de duas estruturas de dados básicas, uma para registrar a situação da linha de varredura (*sweep line status*), e a outra que registra eventos ocorridos durante a varredura (*event-point schedule*).

A idéia consiste em deslocar uma reta vertical pelo conjunto de segmentos, buscando identificar inversões na ordem em que esta reta encontra dois segmentos quaisquer. Para implementar esta idéia, é necessário definir uma nova relação de comparação, da seguinte forma: considere-se dois segmentos s_1 e s_2 no plano, sendo que s_1 não intercepta s_2. Diz-se que s_1 é *comparável* a s_2 se, para alguma abscissa x, existe uma linha vertical que intercepta tanto s_1 quanto s_2. Assim, diz-se que s_1 está *acima de* s_2 *em* x se, naquela abscissa, a interseção da reta com s_1 está acima da interseção da reta com s_2. Esta relação é denotada como $s_1 >_x s_2$. Na Figura 1.10, temos as seguintes relações: $s_3 >_v s_2$; $s_4 >_v s_3$; $s_4 >_v s_2$; $s_4 >_w s_2$; $s_4 >_w s_3$; $s_2 >_w s_3$.

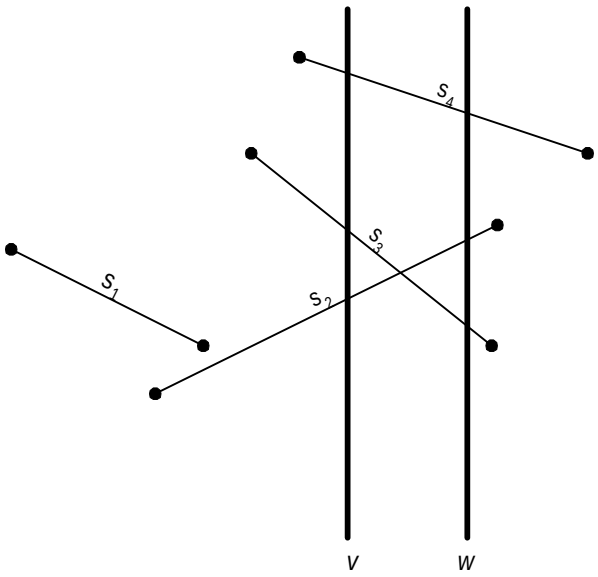


Figura 1.10 - Relação de ordenação entre segmentos

Com esta relação é construída uma ordenação total dos segmentos, que muda à medida em que a linha é deslocada da esquerda para a direita. Nesse processo de varredura do plano, três coisas podem ocorrer:

- o ponto extremo à esquerda de um segmento é encontrado; o segmento é, portanto, inserido na ordenação;

- o ponto extremo à direita de um segmento é encontrado; o segmento é, portanto, retirado da ordenação;

- um ponto de interseção entre dois segmentos s_1 e s_2 foi encontrado; portanto, s_1 e s_2 trocam de posição na ordenação.

Observe-se que, para que s_1 e s_2 possam trocar de posição, é necessário que exista algum x para o qual s_1 e s_2 são consecutivos na ordenação. O algoritmo usa este fato, testando apenas elementos consecutivos, à medida em que novos eventos vão sendo detectados conforme descrito acima.

Portanto, é necessário operar duas estruturas de dados no processo. A primeira (*sweep line status*) é a responsável por manter a ordenação das interseções dos segmentos com a linha de varredura, e é usualmente implementada como um dicionário [PrSh88] ou como uma árvore *red-black* [CLR90]. As operações que o *sweep line status* deve suportar são inserção (`insere`, complexidade $O(\log n)$), exclusão (`exclui`, também $O(\log n)$), e duas funções para determinar qual segmento está imediatamente acima e imediatamente abaixo de um segmento dado na ordenação (`acima` e `abaixo`, $O(1)$). A segunda estrutura de dados (*event-point schedule*) é responsável por manter a seqüência das abscissas que serão analisadas pela linha de varredura, e é implementada como uma fila de prioridades. Deve suportar as clássicas operações de inclusão (`insere`), retirada do elemento de mais alta prioridade (`min`) e uma função que testa a presença de um determinado elemento na estrutura (`membro`), todas com complexidade $O(\log n)$.

Inicialmente, as abscissas dos pontos extremos dos segmentos são ordenadas e inseridas no *event-point schedule*. Em seguida, as abscissas são retiradas a partir da menor, e são realizadas as seguintes operações:

- Se a abscissa corresponder a um ponto extremo à esquerda de algum segmento, inserir o segmento no *sweep line status*. Verificar se existem interseções entre este segmento e os segmentos que estão imediatamente acima e abaixo dele na linha de varredura. Caso exista interseção, a abscissa do ponto de interseção deve ser calculada e inserida no *event-point schedule*, caso já não pertença a ele.

- Se for um ponto extremo à direita, excluir o segmento do *sweep line status*. Verificar se existem interseções entre os segmentos que estão imediatamente acima e abaixo dele na linha de varredura. Caso exista interseção (que estará necessariamente à direita do ponto extremo), a abscissa do ponto de interseção deve ser calculada e inserida no *event-point schedule*, caso já não pertença a ele.

- Se for um ponto de interseção entre dois segmentos, trocar a posição destes segmentos no *sweep line status*. Informar a existência de um ponto de interseção e suas coordenadas.

O algoritmo final está delineado no Programa 1.14. Para melhorar a legibilidade, foram omitidos detalhes da implementação e uso das estruturas de dados básicas, como as listas e a fila de prioridades.

```
procedimento interseçãoNSegmentos
início
     FILA A;
     FILA_DE_PRIORIDADES E;
     SWEEP_LINE_STATUS L;
     Segmento s, s1, s2, s3, s4;
     Ponto I;

     ordenar os 2N pontos extremos por x e y;
     organizar os pontos extremos em uma fila de prioridades E;
     A = nil;
     enquanto (E != nil) faça início
         p = min(E);
         se (p é extremo à esquerda) então início
             s = segmento do qual p é ponto extremo;
             insere(s, L);
             s1 = acima(s, L);
             s2 = abaixo(s, L);
             se (seInterceptam(s1.p1, s1.p2, s.p1, s.p2)) então
                 insere(s1, s, A);
             se (seInterceptam(s2.p1, s2.p2, s.p1, s.p2)) então
                 insere(s, s2, A);
         fim
         senão início
             se (p é extremo à direita) então início
                 s1 = acima(s, L);
                 s2 = abaixo(s, L);
                 se (pontoInterseção(s1.p1,s1.p2, s2.p1,s2.p2, I)) então
                     se (I.x > p.x) então insere(s1, s2, A);
                 exclui(s, L);
             fim
             senão início /* p é uma interseção */
                 s1 = segmento que intercepta s2 em p;
                 s2 = segmento que intercepta s1 em p;
                 /* sendo s1 acima de s2 à esquerda de p */
                 s3 = acima(s1, L);
                 s4 = abaixo(s2, L);
                 se (seInterceptam(s3.p1, s3.p2, s2.p1, s2.p2)) então
                     insere(s3, s2, A);
                 se (seInterceptam(s1.p1, s1.p2, s4.p1, s4.p2)) então
                     insere(s1, s4, A);
                 trocar s1 e s2 de posição em L;
             fim;
         fim;
         /* processamento das interseções */
         enquanto (A != nil) faça início
             retira(s, s1, A);
             x = abscissa da interseção de s e s1;
             se (membro(x, E) = FALSO) então início
                 saída(”Existe interseção entre “,s,” e “, s1);
                 insere(x, E);
             fim;
         fim;
     fim;
fim.
```

Programa 1.14 - Interseção entre N segmentos

Com relação à complexidade deste algoritmo, observa-se que, inicialmente, a operação de ordenação consome tempo $O(n \log n)$. As operações executadas a cada passo no *event-point schedule* consomem $O(\log n)$, correspondente ao pior caso das três operações mutuamente exclusivas que atuam sobre o *sweep line status*. O teste de interseção propriamente dito (`pontoInterseção`) é executado em tempo constante. O número de iterações do laço principal é $2n + K$, onde K é o número de interseções, o que corresponde ao número de eventos que são inseridos no *event-point schedule*. O laço mais interno, de tratamento das interseções, pode potencialmente ser executado a cada iteração do laço principal, sendo portanto executado $O(n + K)$ vezes. Como cada execução do teste de presença na fila de prioridades é executada em tempo logarítmico, temos o tempo $O(\log(n + K))$. Mas como $K \leq \binom{n}{2} = O(n^2)$, então o tempo fica sendo simplesmente $O(\log(n))$. Portanto, o tempo total do laço externo é $O((n + K) \log n)$ [PrSh88].

Com ligeiras modificações, o programa acima pode ser modificado para verificar se existe *alguma* interseção entre os n segmentos. Neste caso, não é necessário manter a fila de prioridades, pois a primeira interseção detectada interromperá o algoritmo, e o *event-point schedule* pode ser implementado como um simples arranjo [CLR90]. Com isso, a complexidade computacional cai para $O(n \log n)$, tendo sido demonstrado que este resultado é ótimo [PrSh88].

Em SIG, o algoritmo de detecção de interseções entre n segmentos tem muita utilidade na dedução de relações topológicas (como *toca* ou *cruza*), na detecção de interseções entre poligonais, entre polígonos ou entre poligonais e polígonos. Também serve para verificar a qualidade de dados digitalizados, testando se uma dada poligonal ou polígono possui auto-interseções indesejáveis.

1.1.4 Simplificação de poligonais

Muitas entidades do mundo real podem ser modeladas como linhas ou, mais genericamente, poligonais. Essas entidades são freqüentes em bases de dados geográficas, onde correspondem tipicamente a cerca de 80% do volume de dados vetoriais [McSh92]. São usadas para representar feições tais como rios, estradas, ruas, linhas de transmissão e adutoras. Os nomes dados pelos SIG comerciais a essas entidades, no entanto, variam muito: linha, polilinha (*polyline*), *line string*, arco, *1-cell*, poligonal, cadeia (*chain*), e outros [Davi97]. A complexidade das representações lineares em SIG pode variar de simples segmentos de reta (dois pares de coordenadas), como um trecho de tubulação de esgoto, até poligonais contendo milhares de pares de coordenadas, como um rio ou uma curva de nível.

Os algoritmos que trabalham com poligonais[3] são muito importantes para os SIG, uma vez que diversas operações básicas, freqüentemente repetidas, são baseadas neles. Em particular, estamos interessados em estudar problemas relacionados à representação de objetos utilizando poligonais, visando conseguir formas de representação mais simples e

[3] Deste ponto em diante, será utilizado o termo *poligonal*, em lugar de simplesmente *linha*, para evitar confusão com a definição geométrica da linha reta (infinita).

compactas a partir de dados mais detalhados. Dentro desse escopo, é necessário levar em consideração que uma das características da poligonal em cartografia é o fato de possuir sempre uma espessura [Peuc75][Bear91], o que a distingue da linha geométrica ideal.

1.1.4.1 Caracterização do Problema

Linhas poligonais são utilizadas em muitas situações para aproximar e representar vetorialmente os limites de objetos complexos encontrados em aplicações de cartografia, SIG, computação gráfica, reconhecimento de padrões e outros [ImIr86]. O problema de simplificação de linhas é particularmente importante em cartografia e SIG, e é estudado intensivamente desde os anos 60, quando ocorreram as primeiras experiências com o uso de instrumentos de transcrição de mapas para o computador, como a mesa digitalizadora. No processo de digitalização de linhas com esses instrumentos, freqüentemente são introduzidos vértices em excesso, vértices que, se descartados, não provocariam uma alteração visual perceptível na poligonal. Assim, um primeiro objetivo para algoritmos de simplificação de linhas é "limpar" (significativamente, o verbo utilizado em inglês é *weed*, "capinar") a poligonal de pontos claramente desnecessários, do ponto de vista de sua visualização [Weib95], mantendo a qualidade de sua aparência gráfica.

Outro objetivo é o de gerar uma nova versão da linha, uma versão mais adequada para a representação do mesmo fenômeno geográfico em outra escala, menor que a escala original de digitalização. Neste caso, está sendo obtida uma *generalização* da linha [McSh92]. Em uma extensão deste enfoque, existe o interesse em organizar os vértices da poligonal de tal forma que seja possível produzir, dinamicamente, versões generalizadas adequadas para uma escala definida no momento da visualização [Oost93][OoSc95], conseguindo portanto gerar múltiplas representações geométricas para o mesmo fenômeno sem introduzir dados redundantes. No entanto, a utilização de métodos e algoritmos desenvolvidos originalmente apenas pensando na redução do número de vértices da linha podem não ser adequados para alcançar o objetivo de generalização [LiOp92], em geral por não conseguirem uma boa representação geométrica[4], e portanto devem ser analisados cuidadosamente quanto a este aspecto.

Medidas de proximidade. Assim, o problema de simplificação de linhas consiste em obter uma representação *mais grosseira* (formada por *menos vértices*, e portanto *mais compacta*) de uma poligonal a partir de uma representação mais refinada, atendendo a alguma restrição de aproximação entre as duas representações. Essa restrição pode ser definida de várias maneiras [McMa86], mas é em geral alguma medida da proximidade geométrica entre as poligonais, tais como o máximo deslocamento perpendicular permitido (Figura 1.11a) ou o mínimo deslocamento angular permitido (Figura 1.11b). Na Figura 1.11a, o vértice 2 será mantido, uma vez que a distância entre ele e a reta que passa pelos vértices 1 e 3 é superior à permitida. Na Figura 1.11b, o vértice 3 será eliminado, uma vez que o ângulo $3\hat{2}4$ é menor que o mínimo tolerável. Uma alternativa

[4] Para auxiliar na manutenção do aspecto natural da poligonal, existem enfoques que integram algoritmos de simplificação com algoritmos de suavização [McMa89].

mais rara é a área entre as poligonais (Figura 1.11c), onde se estabelece um limite para ao deslocamento de área.

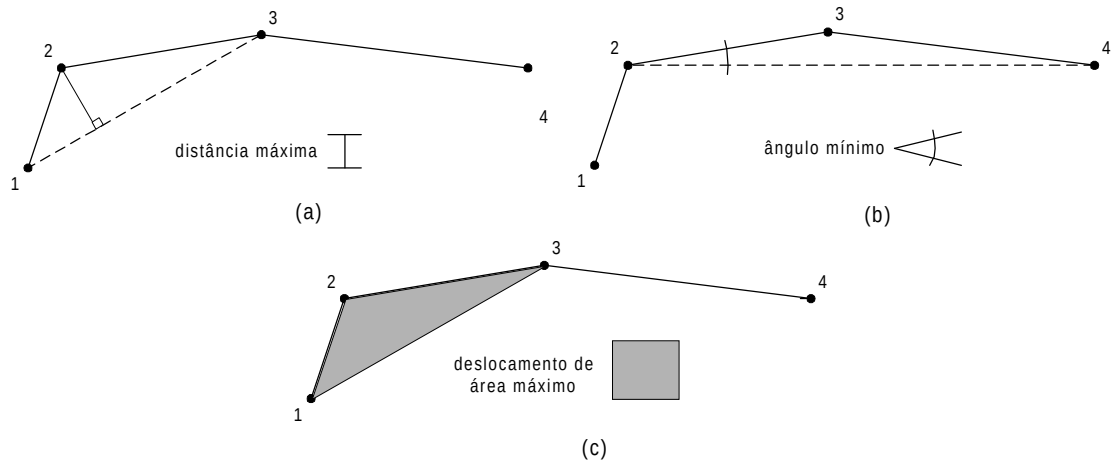


Figura 1.11 - Medidas de proximidade para simplificação de linhas

Dentre todas as medidas possíveis, a mais utilizada é a distância perpendicular. Este fato provavelmente deriva de trabalhos antigos, como o de Perkal ([Perk66] apud [Bear91]). Perkal propôs o conceito de *banda epsilon* como sendo a região ao redor da poligonal que contém todos os pontos do plano situados a uma distância menor que ou igual a ε, em uma tentativa de simular o comportamento da linha cartográfica, que tem largura definida [Peuc75]. Foi também definido que uma poligonal é *ε-convexa* se todos os pontos dela tivessem raio de curvatura superior a ε. Caso isso não ocorra, a banda epsilon se auto-intercepta, indicando perda de legibilidade. Este raciocínio valida o trabalho com distâncias perpendiculares, embora nos algoritmos que a utilizam não esteja explícito qualquer teste de ε-convexidade[5].

O conceito de banda de tolerância, apoiado no cálculo de distâncias perpendiculares, é utilizado em grande parte dos algoritmos de simplificação que serão apresentados a seguir. Um problema eventualmente abordado na literatura é a escolha do parâmetro de tolerância (ε), e sua correlação com a escala da representação simplificada.

Uma regra freqüentemente utilizada em cartografia é a chamada *Lei do Radical* [ToPi66], que determina que o número de objetos a serem mantidos em uma operação de generalização deve ser proporcional à raiz quadrada da mudança de escala. Esta regra foi deduzida a partir da observação empírica da tendência apresentada pelos cartógrafos em manter aproximadamente mesma quantidade de objetos em um mapa de determinada escala. Considerando seu sucesso para esta finalidade prática, foi tentada sua adaptação para determinar a variação da tolerância em função da escala, e para definir o número de segmentos a manter em cada poligonal simplificada. No entanto,

[5] Um algoritmo baseado nestes conceitos foi implementado em um software chamado `WHIRLPOOL` [Bear91], mas sua utilização é comprometida por sua tendência em alterar a topologia percebida das poligonais durante o processo de generalização, especialmente nas situações em que canais estreitos e penínsulas são estrangulados e "ilhas" inexistentes são formadas. Devido a este problema, o algoritmo de Perkal não será abordado por este trabalho.

seu efeito é inócuo para o problema de simplificação, pois conduz a uma seleção aleatória dos objetos e segmentos de poligonal que serão mantidos, assumindo que a poligonal é um conjunto de vértices equiprováveis [Butt85]. Assim, em generalização a Lei do Radical continua sendo útil na determinação prática de *quantos* objetos devem ser mantidos pelo processo de generalização – embora não permita determinar *quais* seriam estes objetos.

Um enfoque mais interessante é o que determina a tolerância com base no tamanho do menor objeto visível em uma determinada escala [LiOp92]. Este tamanho pode ser dado em termos de uma distância medida no espaço de coordenadas do mapa plotado, ou seja, em milímetros do papel, independente da escala utilizada. Assim, é definida uma correspondência linear entre a escala e a tolerância linear adotada. Não existe, contudo, consenso sobre este critério. Existem indicações que o valor ideal seria função não apenas da escala, mas também da complexidade da poligonal [Horn85][Butt89]. Por exemplo, um parâmetro fixo poderia simplificar suficientemente uma poligonal mais simples, e não simplificar suficientemente uma poligonal mais complexa. Este fenômeno pode ocorrer até mesmo dentro de uma dada poligonal, em situações da natureza que fazem com que a estrutura do fenômeno representado pela poligonal mude.

Apesar de todos os problemas relatados, a escolha de um parâmetro fixo de tolerância parece ser mais indicado para aplicações práticas do que, por exemplo, a minimização do número de segmentos da poligonal [GHMS93], ou o acoplamento da distância linear com algum critério de otimização geométrica [Crom88][CrCa91]. A escolha do parâmetro de tolerância linear ideal é ainda discutida, não havendo consenso na literatura. Para aplicações práticas, no entanto, vai-se levar em especial consideração as necessidades da aplicação proposta. Portanto, a escolha do parâmetro de tolerância, seja ele linear, angular ou de área, buscará eficiência geométrica e computacional na generalização de poligonais para representação em tela.

Cálculo de distâncias ponto-reta. Grande parte dos algoritmos de simplificação que serão apresentados a seguir necessita realizar de maneira eficiente cálculos de distância entre um ponto dado e uma reta definida por outros dois pontos. A maneira mais interessante de calcular essa distância é utilizar o produto vetorial, conforme apresentado na seção 1.1.2, para determinar a área *S* do triângulo formado por um ponto *A* e uma reta definida por outros dois (*B* e *C*), de acordo com a equação 1.1. Assim, a distância do ponto *A* à reta definida pelos pontos *B* e *C* pode ser calculada como:

$$d = \frac{|S|}{dist(B,C)}$$

onde *dist(B, C)* é a distância euclidiana entre os pontos B e C, e o único valor que tem que ser testado contra zero para evitar erros numéricos no processamento.

Algoritmos hierárquicos e não-hierárquicos. O resultado do algoritmo de simplificação pode ser (1) uma nova poligonal formada por um subconjunto dos pontos da poligonal original, ou (2) uma poligonal que é formada por pontos distintos dos que formam a poligonal original, à exceção do primeiro e do último [ZhSa97]. No primeiro caso, se a aplicação de tolerâncias progressivamente menores simplesmente causam a inclusão de novos vértices à nova poligonal, o algoritmo é dito *hierárquico* [Crom91]. Mais formalmente, o algoritmo hierárquico é aquele em que todos os vértices

selecionados para produzir uma poligonal de n vértices serão também selecionados quando for gerada uma poligonal com $n+1$ vértices.

Na literatura existem propostas de utilização de algoritmos hierárquicos para construir bases de dados geográficas independentes de escala [BCA95][OoSc95]. Estes estudos indicam que os algoritmos hierárquicos são mais eficientes na redução do tempo operacional, já que a simplificação fica reduzida a uma operação de recuperação de dados ou, caso tenha sido formada uma estrutura de dados adequada, a uma operação de *pruning* [Crom91]. Por outro lado, os algoritmos não-hierárquicos tendem a produzir representações mais eficientes, com relação à preservação de algumas características geométricas da linha [BCA95], de acordo com parâmetros geométricos estabelecidos na literatura (vide seção 0).

Classificação dos algoritmos. Uma classificação dos algoritmos de simplificação de linhas foi proposta em [McMa87a], considerando a porção da linha que é processada a cada passo (Tabela 1.3).

Tabela 1.3 - Classificação dos algoritmos de simplificação de poligonais

Categoria	**Descrição**	**Exemplos**
Algoritmos de pontos independentes	Não consideram as relações geométricas entre vértices vizinhos; operam de forma independente da topologia	*k*-ésimo ponto [Tobl64] seleção aleatória de pontos [RSM78]
Algoritmos de processamento local	Usam as características dos vértices vizinhos imediatos para determinar seleção/rejeição do ponto	Jenks [Jenk81] Visvalingam-Whyatt [ViWh93]
Algoritmos de processamento local restrito estendidos	Pesquisam além dos vizinhos imediatos, avaliando seções da poligonal de cada vez. O tamanho das seções depende de critérios baseados em distâncias, ângulos ou número de vértices	Lang [Lang69] Opheim [Ophe81]
Algoritmos de processamento local estendido irrestrito	Pesquisam além dos vizinhos imediatos, avaliando seções da poligonal de cada vez. O tamanho das seções é limitado pela complexidade geomorfológica da poligonal, e não por critérios determinados no algoritmo	Reumann-Witkam [ReWi74] Zhao-Saalfeld [ZhSa97]
Algoritmos globais	Consideram a poligonal inteira no processamento. Selecionam pontos críticos iterativamente.	Douglas-Peucker [DoPe73]

A classificação acima não considera as propostas incluídas em [PAF95], onde são apresentadas duas novas formas de representação simplificada de poligonais. A primeira delas é a representação freqüencial, baseada em séries de Fourier e *wavelets*, que tentam capturar as tendências oscilatórias presentes em alguns tipos de linhas, como curvas de nível e hidrografia. A segunda é a representação da poligonal com uma seqüência de

curvas algébricas (conjuntos de arcos cúbicos), adequada para estradas e outras feições construídas pelo homem. Esta adoção de recursos diferentes para a simplificação de elementos diferentes conduz à necessidade de dividir a linha em seções que sejam razoavelmente homogêneas em relação a alguns parâmetros geométricos, tais como sinuosidade, complexidade, homogeneidade local e densidade de pontos. De modo geral, no entanto, os algoritmos de simplificação existentes não conseguem captar este tipo de comportamento da linha, com a possível exceção dos algoritmos globais, e ainda assim com problemas [ViWh93].

Avaliação da qualidade da simplificação. A avaliação da qualidade da simplificação foi proposta por McMaster [McMa86] com base em uma série de medidas geométricas. Estas medidas são divididas em duas categorias: medidas de atributos de uma única linha, e medidas de deslocamento entre a poligonal original e a poligonal resultante. São ainda divididas em grupos, de acordo com a grandeza geométrica que está sendo avaliada em cada caso. Estas medidas estão listadas na Tabela 1.4.

Tabela 1.4 - Medidas para avaliação da qualidade da simplificação de linhas

I. Medidas de atributos lineares	**A. Dados sobre o comprimento**	1. Razão de mudança no comprimento da linha
	B. Dados sobre vértices	2. Razão de mudança no número de vértices 3. Diferença do número médio de vértices por unidade de comprimento 4. Razão de mudança do desvio padrão de número de vértices por unidade de comprimento
	C. Dados sobre ângulos	5. Razão de mudança da angularidade (somatório dos ângulos entre vetores consecutivos) 6. Razão de mudança da angularidade à esquerda (positiva) 7. Razão de mudança da angularidade à direita (negativa) 8. Diferença na mudança angular média por unidade de comprimento 9. Diferença na mudança angular média para cada ângulo individual 10. Razão de mudança do número de ângulos positivos 11. Razão de mudança do número de ângulos negativos 12. Diferença na mudança angular positiva média para cada ângulo individual 13. Diferença na mudança angular negativa média para cada ângulo individual
	D. Dados sobre curvilinearidade	14. Razão de mudança do número de segmentos curvilíneos (seqüências de ângulos positivos ou negativos) 15. Razão de mudança da média do número de segmentos curvilíneos 16. Razão de mudança do comprimento médio dos segmentos curvilíneos 17. Razão de mudança do desvio padrão do comprimento médio dos segmentos curvilíneos
II. Medidas de deslocamento linear	**E. Dados de diferenças vetoriais**	18. Somatório das diferenças vetoriais por unidade de comprimento 19. Número de diferenças vetoriais positivas por unidade de comprimento 20. Número de diferenças vetoriais negativas por unidade de comprimento 21. Somatório das diferenças vetoriais positivas por unidade de comprimento 22. Somatório das diferenças vetoriais negativas por unidade de comprimento
	F. Dados de diferenças poligonais	23. Diferença de área total (área entre as poligonais)

		24. Número de polígonos-diferença positivos por unidade de comprimento 25. Número de polígonos-diferença negativos por unidade de comprimento 26. Diferença de área positiva por unidade de comprimento 27. Diferença de área negativa por unidade de comprimento
	G. Dados de perímetro	28. Perímetro total das diferenças de área (comprimento ao redor dos polígonos de diferença) por unidade de comprimento 29. Perímetro total das diferenças de área positivas (comprimento ao redor dos polígonos de diferença) por unidade de comprimento 30. Perímetro total das diferenças de área negativas (comprimento ao redor dos polígonos de diferença) por unidade de comprimento

Foram analisadas em [McMa86] todas as 30 medidas listadas na Tabela 1.4, para 31 poligonais cartográficas diferentes[6], com características geomorfológicas variadas. As poligonais foram inicialmente digitalizadas e "limpas", até que se tornassem, quando plotadas, cópias fiéis das linhas contidas nos mapas originais. Em seguida, o mesmo algoritmo de simplificação (no caso, o algoritmo Douglas-Peucker) foi aplicado a cada uma delas, com várias tolerâncias, obtendo-se as medidas acima para cada situação. Ao final do processo, após uma análise estatística rigorosa, o autor concluiu pela possibilidade de se reduzir o número de medições a apenas seis, que são suficientemente descorrelacionadas para permitir uma comparação adequada. Estas medidas são:

Mudança percentual no número de vértices: fornece uma indicação do grau de compactação atingido, mas só pode ser usada para verificar a qualidade da simplificação em conjunto com outras medidas. Mais formalmente, pode ser definida como:

$$MPCV = \frac{n'}{n} \times 100$$

onde n' é o número de vértices da poligonal simplificada, e n é o número de vértices da poligonal original.

Mudança percentual no desvio padrão do número de coordenadas por unidade de comprimento: mede a regularidade da inserção de vértices ao longo da poligonal, indicando se a linha resultante assumiu uma densidade uniforme de vértices em relação à linha original. Ou seja:

$$MPCN = \frac{\sigma(n')/l'}{\sigma(n)/l} \times 100$$

onde l' é o comprimento da poligonal simplificada, e l é o comprimento da poligonal original.

[6] Neste e em outros estudos existe a preocupação em utilizar *naturally occurring lines*, ou seja, poligonais usadas para delimitar ou representar fenômenos que ocorrem na natureza, considerando uma ampla gama de fenômenos: curso e estuário de rios, linhas costeiras e curvas de nível, por exemplo. Dentro de cada tipo de fenômeno, busca-se selecionar poligonais com comportamento variado. Por exemplo seriam incluídos tanto rios jovens, com muitos meandros de curvatura brusca, quanto rios antigos, com curvas suaves e em menor quantidade. Um estudo anterior que procurou utilizar estes critérios foi [Mari79].

Mudança percentual na angularidade: avalia a redução da "microssinuosidade" após a simplificação. Esta medida pode ser definida como:

$$MPCA = \frac{\sum_{i=0}^{n'-3} ang(v_i, v_{i+1}, v_{i+2})}{\sum_{i=0}^{n-3} ang(v_i, v_{i+1}, v_{i+2})} \times 100$$

onde *ang* é uma função que calcula o ângulo definido por três pontos (Figura 1.11b), que no caso é utilizada em segmentos consecutivos na poligonal simplificada e na poligonal original.

Deslocamento vetorial total por unidade de comprimento: indica o deslocamento geométrico total da linha com relação à original. Ou seja,

$$DVT = \frac{\sum_{i=0}^{n'-1} d(v_i, P)}{l}$$

onde *P* é a poligonal original, e a função *d* calcula a distância perpendicular entre um ponto dado e uma poligonal (Figura 1.11a).

Deslocamento de área total por unidade de comprimento: como a anterior, indica o deslocamento geométrico total da linha, só que considerando a área entre a poligonal simplificada e a original:

$$DAT = \frac{A(P', P)}{l}$$

onde *A* é uma função que calcula a área total entre duas poligonais (Figura 1.11c), e *l* é o comprimento da poligonal original.

Mudança percentual do número de segmentos curvilíneos: quando processos de suavização são aplicados juntamente com a simplificação, muitos segmentos curvilíneos poderão ser eliminados. Cada segmento curvilíneo é caracterizado por uma seqüência de mudanças angulares (ângulos entre segmentos consecutivos) à direita ou à esquerda. Esta medida indica o grau de suavização obtido na simplificação, pela eliminação de segmentos curvilíneos, e pode ser definida como:

$$MPCS = \frac{NSC'}{NSC} \times 100$$

onde *NSC'* e *NSC* representam, respectivamente, o número de segmentos curvilíneos na poligonal simplificada e na original.

Com base nas medidas propostas em [McMa86], um estudo posterior [McMa87b] comparou nove algoritmos, chegando à conclusão de que quatro deles apresentavam comportamento superior: Douglas-Peucker, Opheim, Reumann-Witkam e Lang. A avaliação visual, no entanto, indica que o algoritmo Douglas-Peucker se comportou

melhor quanto ao deslocamento de área. No entanto, é o mais computacionalmente complexo dos quatro. O artigo conclui que o algoritmo de Lang é mais adequado para certas tarefas de mapeamento menos exigentes, tais como mapeamento temático. Estes algoritmos, e outros, serão apresentados em detalhes em seguida. Alguns algoritmos de interesse específico deste trabalho, tais como Visvalingam-Whyatt e Zhao-Saalfeld, não fizeram parte do estudo, pois são mais recentes do que ele.

Algoritmos. Existem vários algoritmos na literatura que se propõem a resolver o problema de simplificação de poligonais. Conforme discutido na seção 0, a comparação entre os algoritmos não pode se ater simplesmente à análise de sua complexidade computacional. São muito importantes parâmetros de avaliação da qualidade da representação geométrica resultante [Mari79][McMa86][Whit85], além da avaliação do grau de compactação atingido. Considerando as aplicações em generalização, é também necessário avaliar as possibilidades de geração de estruturas de dados hierárquicas, para possibilitar a produção dinâmica de novas representações.

Em seguida serão apresentados alguns dos algoritmos de simplificação de linhas propostos na literatura[7]. A seleção dos algoritmos foi feita com base na sua importância histórica, no grau de correção cartográfica, em sua eficiência computacional, e também com base na aplicabilidade ao problema de generalização dinâmica. Em todas as descrições abaixo, será utilizada a letra *n* para denotar o número de pontos na linha original, e *n'* indicará o número de pontos na linha simplificada.

1.1.4.2 *k*-ésimo vértice [Tobl64]

Este algoritmo foi proposto por Tobler em 1964, para um experimento de generalização de mapas para a Marinha americana [Tobl64]. É possivelmente o mais antigo algoritmo de simplificação conhecido, e também o mais simples. A idéia é formar a nova poligonal selecionando um vértice a cada *k*, sendo *k* determinado pelo usuário, e desprezando os demais. O grau de generalização obtido é bastante grosseiro, uma vez que o algoritmo não considera qualquer fator geométrico, correndo portanto o risco de descaracterizar completamente a linha.

Observe-se que a mesma lógica é utilizada para simplificar imagens digitais, selecionando arbitrariamente um pixel a cada *k*, em um processo conhecido como *subamostragem* [Jain89], que é um caso particular da reamostragem pelo método do vizinho mais próximo, quando se reduz o tamanho da imagem [Wolb90]. O método tem também pouco sucesso pois, apesar de ser simples, tende a provocar o efeito de *aliasing* sobre a imagem.

Complexidade computacional. O(n/k) = O(n')

1.1.4.3 Vértice aleatório [RSM78]

Proposto por Robinson *et al.* [RSM78], este algoritmo pouco acrescenta ao do *k*-ésimo vértice. Parte do princípio de que a linha cartográfica é composta por vértices

[7] São deixados intencionalmente de fora alguns algoritmos, como os propostos em [Deve85], [Will78] e [Joha74], por não acrescentarem elementos relevantes à discussão.

equiprováveis [Peuc75][Butt85]. Trata-se de selecionar aleatoriamente uma quantidade predeterminada (*n'*) dos vértices que compõem a poligonal, gerando desta forma uma nova poligonal simplificada. Como no caso do *k*-ésimo vértice, o algoritmo não considera a geometria da linha, e também corre o risco de desprezar pontos característicos.

Complexidade computacional. O(n')

1.1.4.4 Jenks [Jenk81]

Este algoritmo considera uma seqüência de três vértices na linha, calculando a distância do vértice intermediário à reta definida pelos outros dois [Jenk81]. Quando esta distância é inferior a uma tolerância dada, o vértice central é eliminado, e o algoritmo é reiniciado com o primeiro, o terceiro e o quarto vértices. Se a distância exceder a tolerância, o segundo vértice é mantido, e o processamento recomeça a partir dele (Figura 1.11a). Apesar de não apresentar um tratamento mais sofisticado da geometria da linha, este algoritmo consegue eliminar os vértices efetivamente desnecessários, ou seja, que estão alinhados com os vértices anterior e posterior, e portanto são geometricamente dispensáveis. Em especial, este comportamento é melhor caracterizado quando a tolerância é bastante pequena.

Uma variação interessante deste algoritmo estabelece, como tolerância, não a distância do vértice central ao segmento formado pelos outros dois, mas sim o ângulo entre os segmentos v_1v_2 e v_1v_3. Quando este ângulo estiver abaixo de um valor dado, os pontos são considerados alinhados e v_2 é descartado, sendo reiniciado o processamento em v_3. Caso contrário, v_2 é mantido, e o processamento recomeça por ele (Figura 1.11b).

De acordo com [McMa87a], Jenks posteriormente propôs uma variação desta rotina angular considerando três valores distintos de tolerância, denominados min_1, min_2 e *ang*. Se a distância entre v_1 e v_2 for inferior a min_1, ou se a distância entre v_1 e v_3 for inferior a min_2, então v_2 é eliminado. Caso as duas distâncias ultrapassem os valores mínimos, é então testado o ângulo entre os segmentos v_1v_2 e v_1v_3 contra a tolerância *ang*, da maneira descrita acima.

*Complexidade computacional***.** O pior caso do algoritmo de Jenks ocorre quando todos os vértices são eliminados. Assim, cada iteração vai testar se o vértice v_i ($2 \le i \le n-2$) vai ser ou não mantido, calculando as distâncias dos vértices entre v_2 e v_i à reta definida por v_1 e v_{i+1}, totalizando portanto i - 2 cálculos de distância. Este comportamento pode ser traduzido pelo seguinte somatório:

$$(n-2)\cdot\sum_{i=2}^{n-1}(i-2) = (n-2)\cdot\left[\frac{(n-2)(n-1)}{2} - 2\cdot(n-2)\right]$$

A complexidade computacional do algoritmo de Jenks no pior caso é, portanto, $O(n^2)$.

O melhor caso, por outro lado, ocorre quando nenhum vértice é eliminado. Nesta situação, cada um dos n - 2 vértices intermediários é testado uma única vez contra seus vizinhos imediatos, produzindo um cálculo de distância ponto-reta. O algoritmo é, assim, $O(n)$ no melhor caso.

Observe-se que este comportamento é o inverso do apresentado pelo algoritmo Douglas-Peucker, que será apresentado adiante, no sentido de que seu pior caso ocorre quando elimina todos os vértices, e o melhor ocorre quando todos os vértices são mantidos. Sendo assim, este algoritmo poderá ser mais eficiente que o Douglas-Peucker em situações de simplificação mínima, e está recomendado para situações em que o interesse seja simplesmente a eliminação de vértices alinhados, com tolerância baixíssima.

1.1.4.5 Reumann-Witkam [ReWi74]

Este algoritmo [ReWi74] utiliza, em cada iteração, duas linhas paralelas a cada segmento da poligonal para determinar uma região de eliminação de vértices. Naturalmente, a tolerância é precisamente a metade da distância entre as paralelas. A poligonal é analisada seqüencialmente, sendo buscado o primeiro segmento que intercepta uma das duas paralelas. Sendo $v_i\ v_{i+1}$ o segmento localizado, o vértice v_i é mantido e todos os intermediários, entre o vértice inicial e v_i, são descartados.

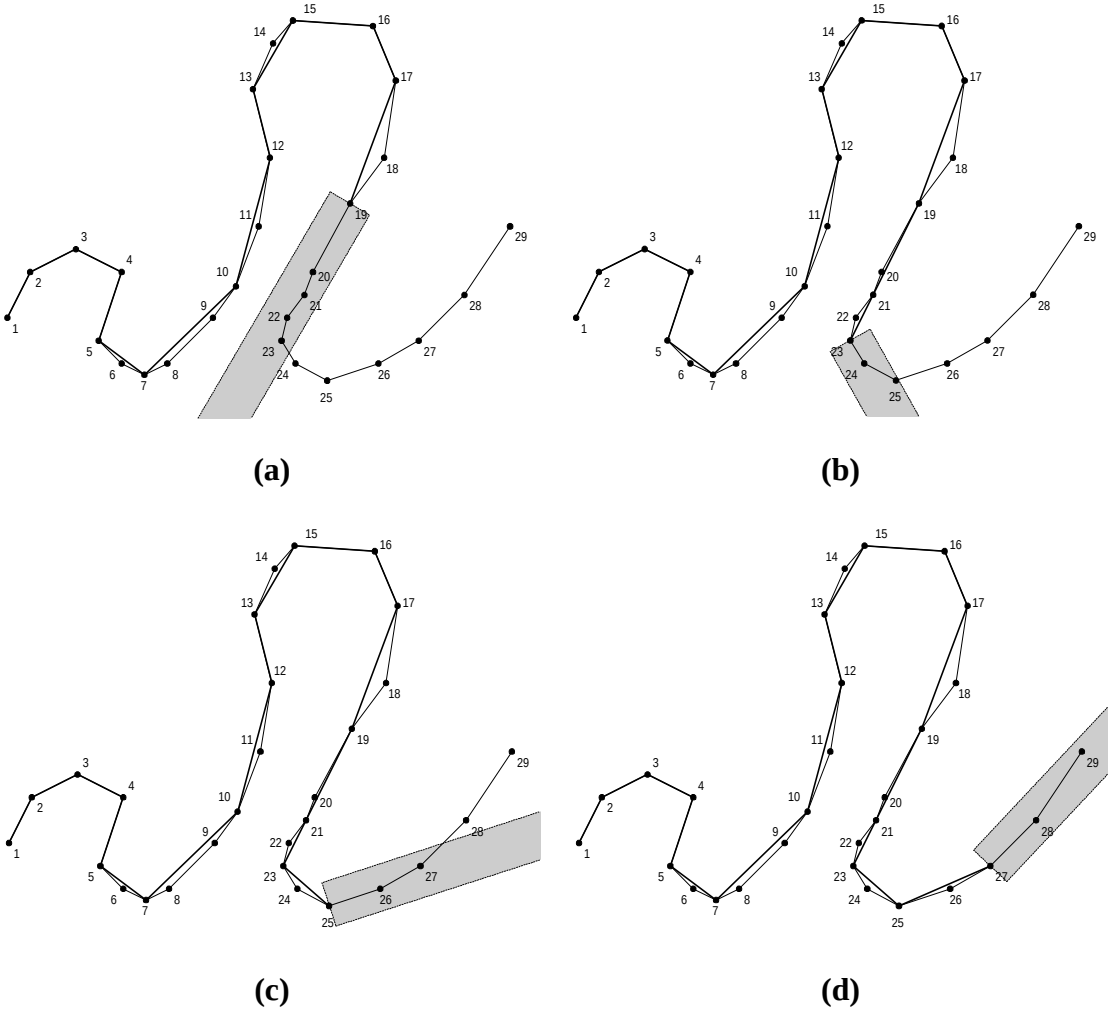


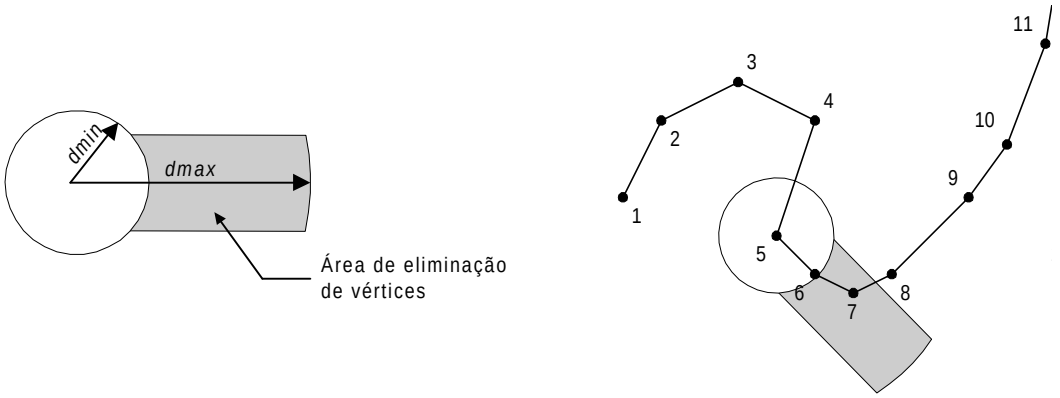
Figura 1.12 - Algoritmo Reumann-Witkam

No exemplo da Figura 1.12, observe-se que, até o vértice v_{19}, poucos vértices haviam sido eliminados. Aplicando as paralelas a partir de v_{19}, na direção do segmento $v_{19}\ v_{20}$, é possível eliminar os vértices v_{20} a v_{22} (Figura 1.12a). O processamento é retomado a

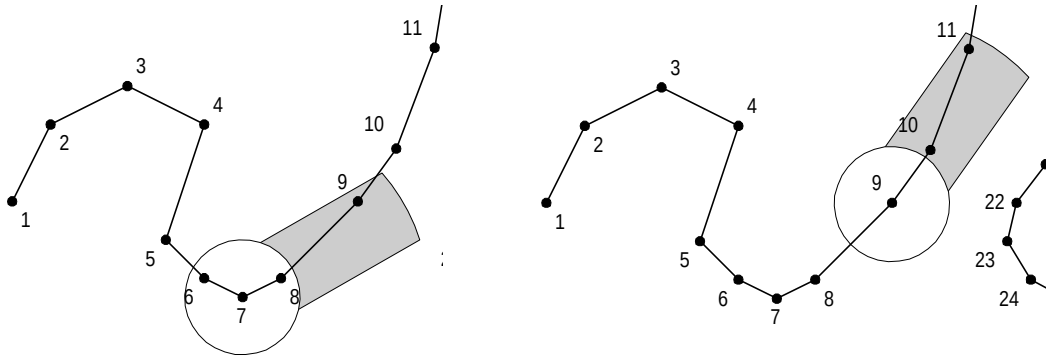
partir do vértice v_{23} (Figura 1.12b), desta vez eliminando o vértice v_{24}. Reiniciando de v_{25}, elimina-se v_{26} (Figura 1.12c). O processamento termina em seguida, eliminando v_{28} e terminando no último vértice, v_{29} (Figura 1.12d). No total, doze vértices foram eliminados.

O algoritmo Reumann-Witkam é, segundo a classificação proposta por McMaster [McMa87a], do tipo *processamento local estendido irrestrito*, pois a verificação é iniciada em um determinado vértice, e prossegue até que algum segmento tenha interceptado uma das paralelas, ou até que a linha termine. Opheim [Ophe81] propôs uma variação deste enfoque em que são impostos limites quanto à seção da linha que é analisada, produzindo assim um algoritmo com *processamento local estendido restrito*. São definidos uma distância mínima (*dmin*) e uma distância máxima (*dmax*), que funcionam limitando, juntamente com as paralelas, uma região que contém todos os vértices que serão eliminados. Como no Reumann-Witkam, o processamento recomeça no primeiro vértice do primeiro segmento que sai da região limitada. Esta variação apresenta a tendência a preservar melhor a aparência das curvas, pois evita que vértices muito próximos e vértices muito distantes do vértice de partida sejam eliminados. Conseqüentemente, o grau de compactação é ainda menor do que o que se pode alcançar com o algoritmo Reumann-Witkam.

Na Figura 1.13a é apresentada a região de eliminação de vértices do algoritmo Opheim. Qualquer vértice que fique dentro da área marcada será eliminado, a menos que seja o primeiro vértice de um segmento que intercepta as linhas paralelas. É o caso do vértice v_7 na Figura 1.13b, que não será eliminado, e servirá de ponto de partida para a próxima iteração. O vértice v_6, também na Figura 1.13b, não será eliminado pois está a uma distância inferior a *dmin* com relação ao vértice de partida, v_5. A mesma situação se repete a partir de v_7, pois v_8 precisa ser mantido e v_9 se torna o novo vértice de partida (Figura 1.13c). Partindo de v_9, finalmente temos a eliminação de v_{10}.



(a) **(b)**



(c) **(d)**

Figura 1.13 - Algoritmo Opheim

*Complexidade computacional***.** Como se pode perceber, cada vértice é considerado apenas uma vez, buscando a interseção de segmentos da poligonal com as retas paralelas. Caso a interseção ocorra, o procedimento é reiniciado a partir do último vértice analisado, mantendo o padrão de varredura. Assim, este algoritmo é claramente linear sobre o número de vértices: $O(n)$. O mesmo se aplica à variação proposta por Opheim.

Apesar de eficiente computacionalmente, seu desempenho cartográfico, por outro lado, deixa bastante a desejar, pois tende a eliminar poucos vértices. Além disso, tende a não eliminar pontos situados em curvas mais abertas, como o vértice v_2 no caso do exemplo da Figura 1.12. Por isso, este algoritmo foi posteriormente modificado por Roberge [Robe85], que o robusteceu matematicamente com testes de linhas críticas verticais e com uma verificação de pontos de inflexão. Roberge propôs também um fator de extensão da linha crítica, visando permitir a eliminação de vértices em curvas de grande raio.

1.1.4.6 Lang [Lang69]

Este algoritmo, proposto por Lang [Lang69], requer dois parâmetros de entrada: uma distância de tolerância, semelhante à do algoritmo de Jenks, e um parâmetro (p) de *look-ahead*, ou seja, uma quantidade de vértices que devem ser considerados a cada etapa do processo. Por exemplo, se p for igual a 5, serão analisados os vértices v_1 a v_6. Nesse caso, será calculada a distância entre os vértices v_2, v_3, v_4 e v_5 à reta que passa por v_1 e v_6. Se a alguma distância obtida for maior que a tolerância, o algoritmo retrocede um ponto (no caso, até v_5) e recomeça, se necessário fazendo o mesmo enquanto existirem vértices intermediários. Se todas as distâncias intermediárias forem menores que a tolerância, os vértices intermediários são eliminados, e o próximo passo será iniciado no último vértice extremo encontrado, considerando novamente p pontos à frente. No exemplo da Figura 1.14, foi necessário recuar de v_5 (Figura 1.14a) até v_3, eliminando apenas v_2 (Figura 1.14d), uma vez que existiam em cada uma das etapas intermediárias (Figura 1.14b e c) vértices fora da faixa de tolerância. O algoritmo recomeça a partir de v_3, o último vértice mantido, e vai analisar em seguida o trecho entre v_3 e v_8.

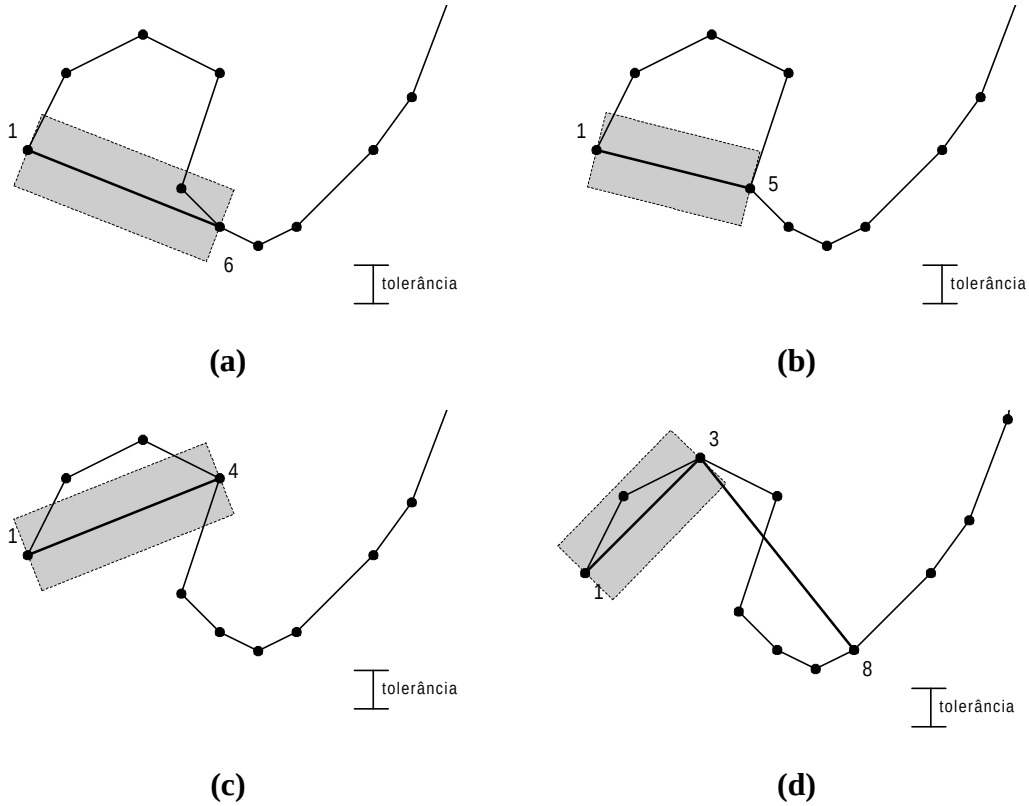


Figura 1.14 - Algoritmo Lang

Este algoritmo se comporta bem do ponto de vista geométrico [McMa87a], sendo capaz de preservar razoavelmente as características da linha original. No entanto, sofre a influência do parâmetro de *look-ahead*: valores diferentes levarão a resultados diferentes. Apesar disso, sua implementação é interessante por fixar a quantidade máxima de pontos que serão analisados a cada etapa, possibilitando um dimensionamento estático de memória em tempo de compilação, caso se limite o valor de *p*.

Com relação ao grau de compactação, observe-se que este algoritmo só será capaz de eliminar $n - n/p$ vértices, uma vez que, em cada intervalo são preservados, no mínimo, os dois vértices extremos. A manutenção dos vértices extremos de cada intervalo prejudica a aparência final da poligonal, pois sua seleção é tão arbitrária quanto a realizada pelo algoritmo do *k*-ésimo vértice.

*Complexidade computacional***.** O pior caso ocorre quando nenhum ponto é eliminado, e portanto o algoritmo precisa fazer o *look-ahead* n/p vezes e retroceder, em cada passo, $p-1$ vezes. Em cada um dos n/p passos, o algoritmo executa $(p-1)(p-2)/2$ cálculos de distância ponto-reta. Assim, o algoritmo é $O(n)$, mas com um fator constante que cresce ao ritmo de $O(p^2)$. Particularmente, quando $n = p$, teremos complexidade $O(n^2)$ para o algoritmo.

No melhor caso, todos os $p-1$ pontos intermediários serão eliminados na primeira passada, e portanto teremos $p-1$ cálculos de distância para cada um dos n/p passos, e portanto o algoritmo é $O(n)$. Observe-se que temos um compromisso entre o tempo de

processamento, dependente de *p*, e o grau de compactação: valores maiores de *p* possibilitarão uma compactação maior, mas provocarão um aumento no custo computacional em relação ao caso linear. Assim, a escolha do valor ideal para *p* é dificultada, e dependerá de uma análise do processo de digitalização com relação à quantidade de vértices desnecessários.

1.1.4.7 Douglas-Peucker [DoPe73]

Este é o mais conhecido e utilizado algoritmo de simplificação de pontos. Foi proposto em 1973 por Douglas e Peucker [DoPe73], e é reconhecidamente o melhor em termos de preservação das características da poligonal original [Mari79][McMa87a], especialmente se utilizado com tolerâncias pequenas [ViWh90]. Curiosamente, o algoritmo foi proposto quase que simultaneamente por Ramer [Rame72] e Duda e Hart [DuHa73], embora visando aplicações diferentes. O algoritmo Douglas-Peucker permanece sendo o mais citado na literatura de geoprocessamento, uma vez que foi originalmente publicado em um periódico da área de cartografia.

```
Procedimento Douglas-Peucker(linha, numvert, tol)

      Procedimento DP(a, f, tol)

      início
            se ((f - a) == 1) então retorne;
            maxd = 0;
            maxp = 0;
            para i = a+1 até f-1 faça
            início
                  d = distância(linha[i], linha[a], linha[f]);
                  se d > maxd então
                  início
                        maxd = d;
                        maxp = i;
                  fim se;
            fim para;
            se maxd > tol então
            início
                  vértice maxp selecionado;
                  DP(a, maxp, tol);
                  DP(maxp, f, tol);
            fim
            senão retorne;
      fim;

início
      vértice 1 selecionado;
      vértice numvert selecionado;
      DP(1, numvert, tol);
fim.
```

Programa 1.15 - Algoritmo Douglas-Peucker

Funcionamento. O algoritmo é recursivo (Programa 1.15), e a cada passo processa o intervalo de pontos contido entre um vértice inicial (chamado de *âncora*) e um vértice final (denominado *flutuante*). É estabelecido um *corredor* de largura igual ao dobro da tolerância, formando duas faixas paralelas ao segmento entre o âncora e o flutuante (Figura 1.16), como no algoritmo de Lang. A seguir, são calculadas as distâncias de todos os pontos intermediários ao segmento básico, ou seja, contidos entre o âncora e o

flutuante. Caso nenhuma das distâncias calculadas ultrapasse a tolerância, ou seja, nenhum vértice fica fora do corredor, então todos os vértices intermediários são descartados. Caso alguma distância seja maior que a tolerância, o vértice mais distante é preservado, e o algoritmo é reiniciado em duas partes: entre o âncora e o vértice mais distante (novo flutuante), e entre o vértice mais distante (novo âncora) e o flutuante. De acordo com este processo, os pontos tidos como críticos para a geometria da linha, a cada passo, são mantidos, enquanto os demais são descartados.

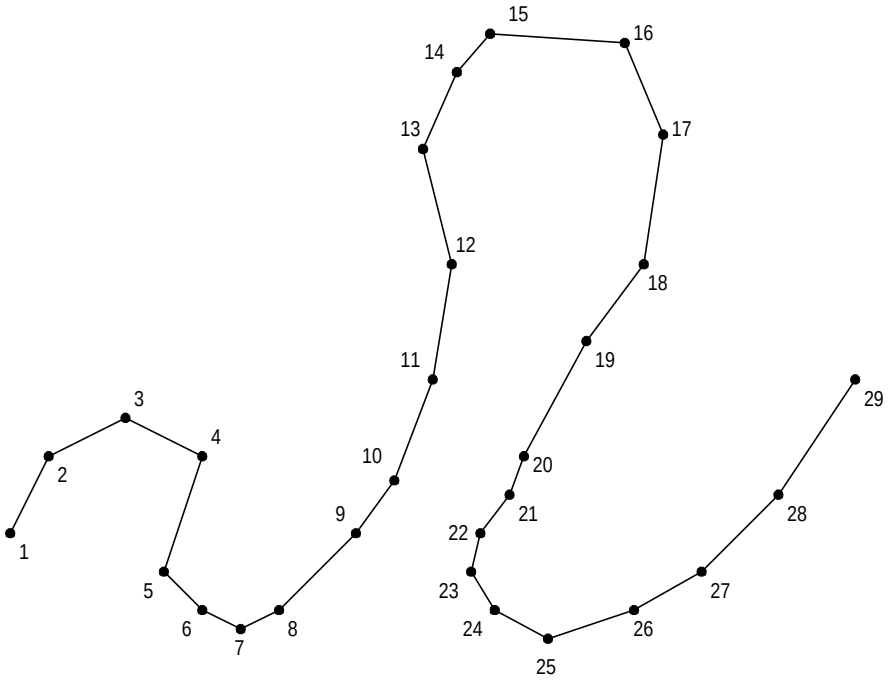


Figura 1.15 - Linha original, 29 vértices

Para a análise deste algoritmo e dos próximos será utilizada a poligonal da Figura 1.15., com 29 vértices. As figuras seguintes ilustram melhor o comportamento do algoritmo Douglas-Peucker. Inicialmente, são calculadas as distâncias dos vértices 2 a 28 até a reta definida pelos vértices 1 e 29. O vértice mais distante nesta primeira iteração é o 15, a uma distância muito superior à tolerância (Figura 1.16). Assim, o vértice 15 é selecionado e o procedimento é chamado recursivamente duas vezes, entre os vértices 1 e 15 e entre os vértices 15 e 29. Continuando pela primeira chamada, o vértice mais distante da reta entre 1 e 15 é o 9, também a uma distância superior à tolerância, e portanto é selecionado (Figura 1.17). Duas novas chamadas recursivas são feitas, e agora estão empilhados os intervalos 1-9, 9-15 e 15-29. No intervalo 1-9, temos também que preservar o vértice 3, e portanto ficamos na pilha com os intervalos 1-3, 3-9, 9-15 e 15-29 (Figura 1.18). Analisando agora o intervalo 1-3, verificamos que o vértice 2 pode ser dispensado (Figura 1.19). Ao final, são preservados os vértices 1, 3, 4, 6, 9, 15, 16, 17, 22, 24, 27 e 29, ou seja, 41% do número original de vértices (Figura 1.20).

A utilização de recursividade no processamento do algoritmo já inspirou implementações paralelas [Mowe96], em que cada nova subdivisão do problema vai sendo atribuída a um processador diferente, sendo gerada ao final uma lista dos vértices selecionados. No entanto, a grande diferença entre o melhor e o pior caso de processamento (vide análise de complexidade computacional) podem ser um problema,

uma vez que poderá haver pouco equilíbrio entre as tarefas atribuídas a cada processador [HeSn92].

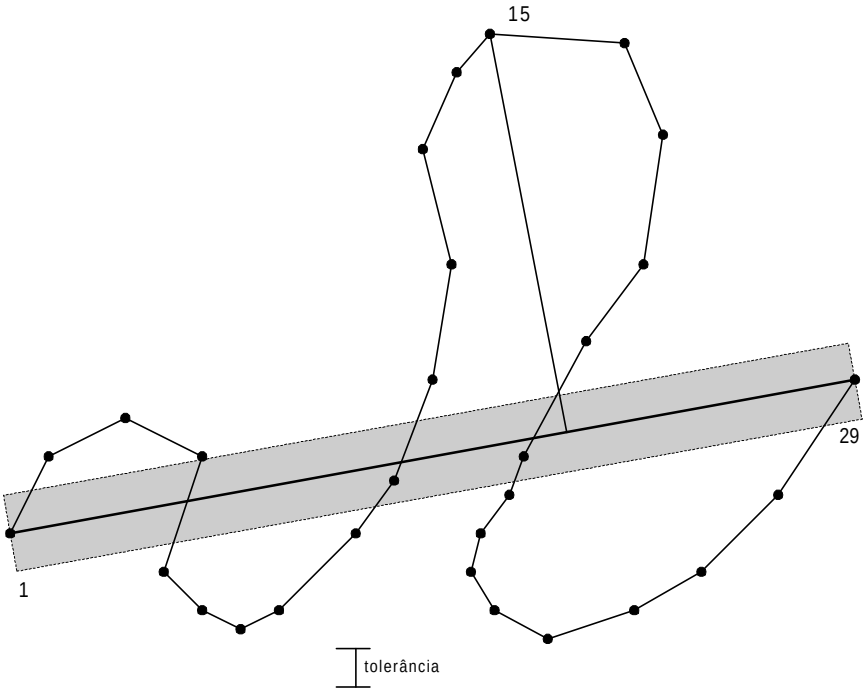


Figura 1.16 - Douglas-Peucker, primeiro passo: seleção do vértice 15

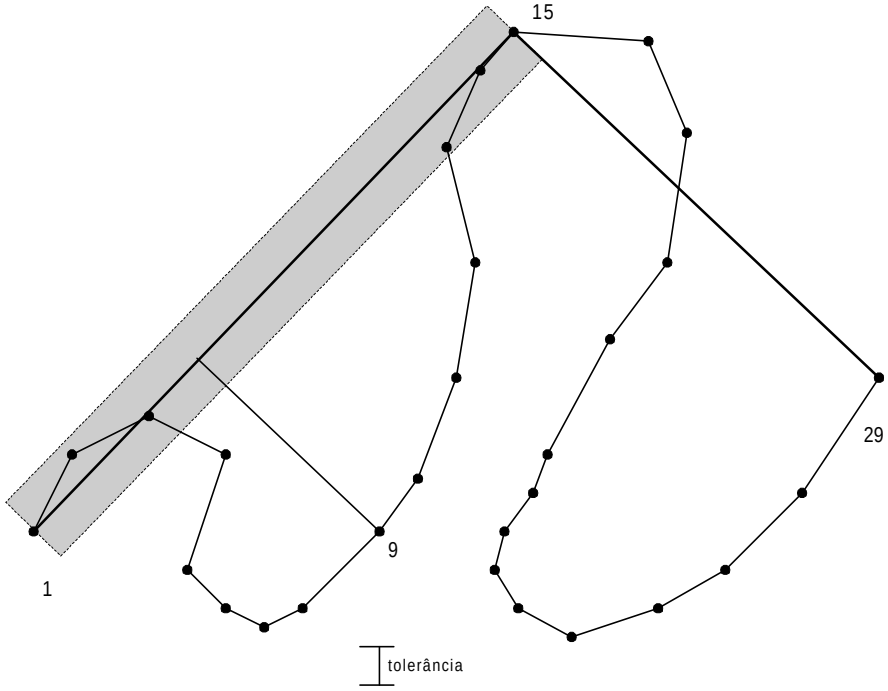


Figura 1.17 - Douglas-Peucker, segundo passo: seleção do vértice 9

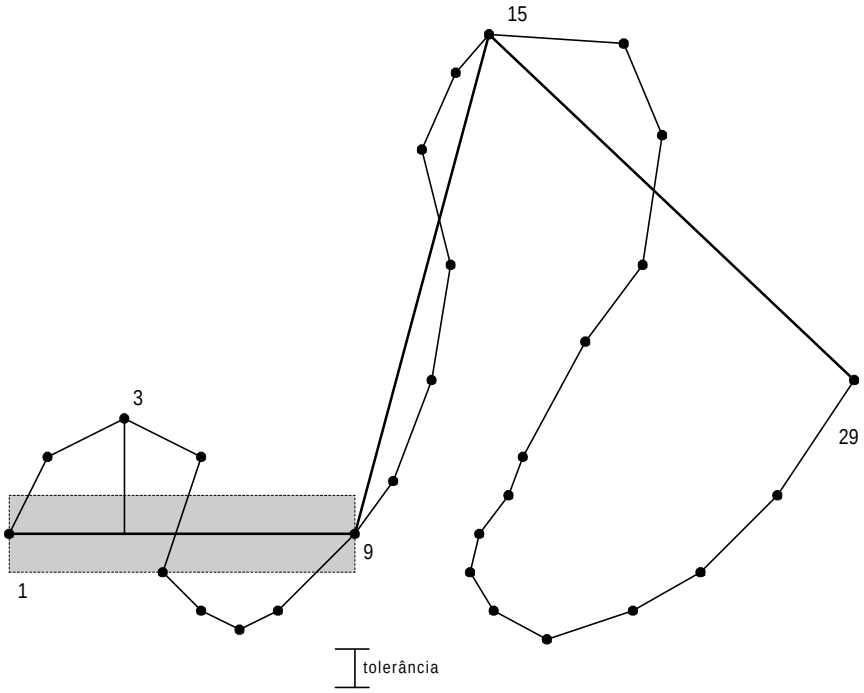


Figura 1.18 - Douglas-Peucker, terceiro passo: seleção do vértice 3

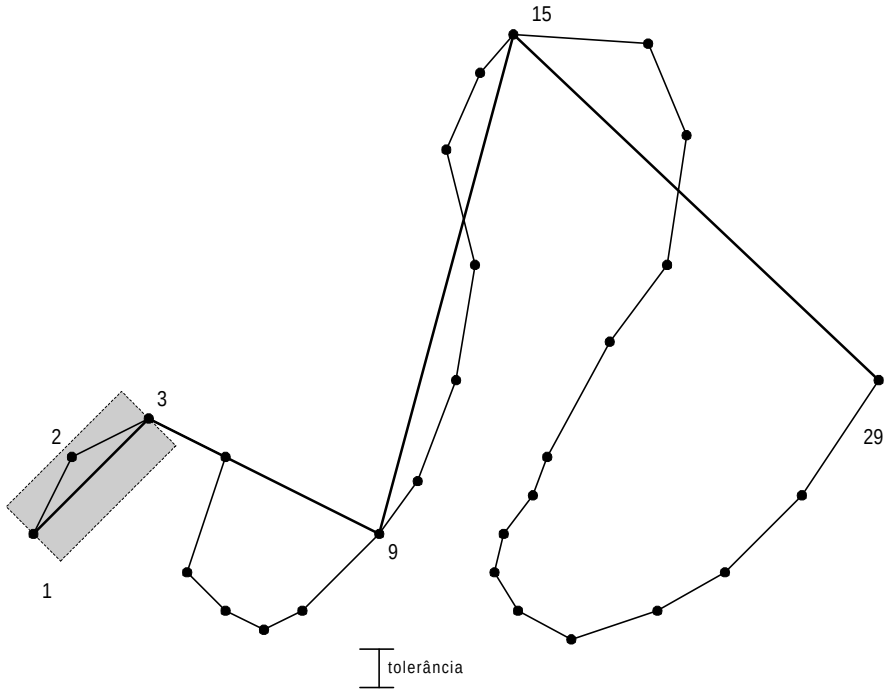


Figura 1.19 - Douglas-Peucker, passo 4: eliminação do vértice 2

O resultado deste algoritmo é aclamado pela literatura como sendo o que mais respeita as características (ou, como no título do artigo de Douglas e Peucker, a “caricatura”) da linha cartográfica [Mari79][Jenk89][McMa87a]. Assim, este algoritmo veio a ser a escolha dos desenvolvedores de software comercial na implementação de funções de simplificação de linhas para processamento pós-digitalização [LiOp92], ou seja, para limpeza de vértices desnecessários. O uso do algoritmo Douglas-Peucker em

generalização, no entanto, é comprometido pelo seu comportamento em situações de generalização mais radical, ou seja, com tolerâncias maiores [ViWi95]. Conforme a situação, o algoritmo pode ser levado a escolher vértices que terminam por deixar a linha com uma aparência pouco natural [ViWh93], com tendência a apresentar “picos” (como no exemplo da Figura 1.21, entre os vértices 17, 24 e 29), com ângulos agudos e mudanças bruscas de direção.

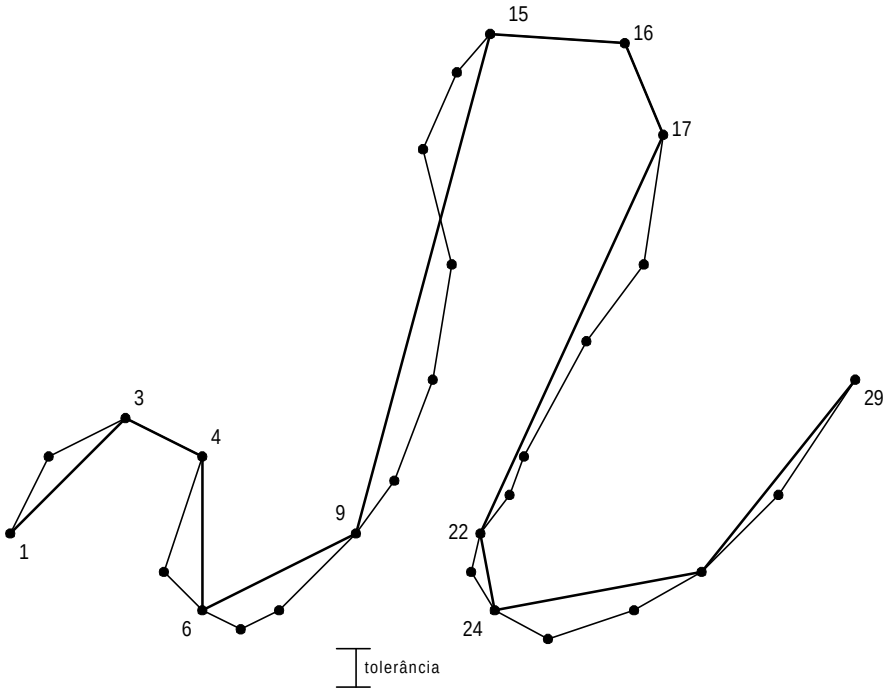


Figura 1.20 - Douglas-Peucker, final

Se a mesma linha da Figura 1.15 for processada novamente com uma tolerância, por exemplo, quatro vezes maior que a apresentada, seriam preservados apenas os vértices 1, 9, 15, 17, 24 e 29, todos pertencentes à solução anterior (Figura 1.21). Portanto, o algoritmo de Douglas-Peucker é hierárquico, pois os pontos são sempre selecionados na mesma ordem, e a tolerância serve para determinar até que ponto o processamento deve ser realizado.

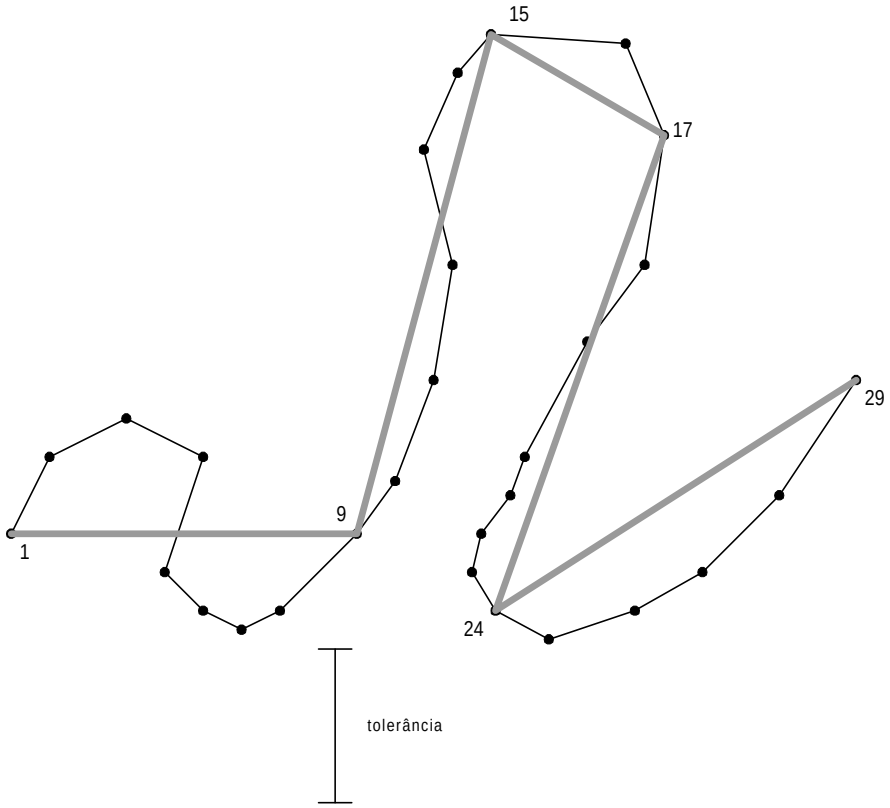


Figura 1.21 - Douglas-Peucker, simplificação radical

Se a tolerância for igual a zero, todos os vértices serão eventualmente selecionados. O armazenamento das subdivisões nos permite representar a hierarquia dos vértices em uma árvore binária [Crom91][Oost93]. Em cada nó desta árvore é representado um vértice selecionado, e é armazenado o valor da distância calculado por ocasião da seleção, que corresponde ao valor `maxd` do Programa 1.15 (Figura 1.22). Tendo sido estabelecido um valor de tolerância, basta caminhar na árvore em preordem para determinar quais vértices serão selecionados. Quando um nó interno contiver um valor de distância inferior à tolerância, o vértice correspondente e todos os descendentes poderão ser eliminados, não sendo necessário continuar com o caminhamento. Observe-se, no entanto, que a árvore binária pode ser bastante desbalanceada, e dificilmente será completa, o que virá a dificultar o seu armazenamento no banco de dados.

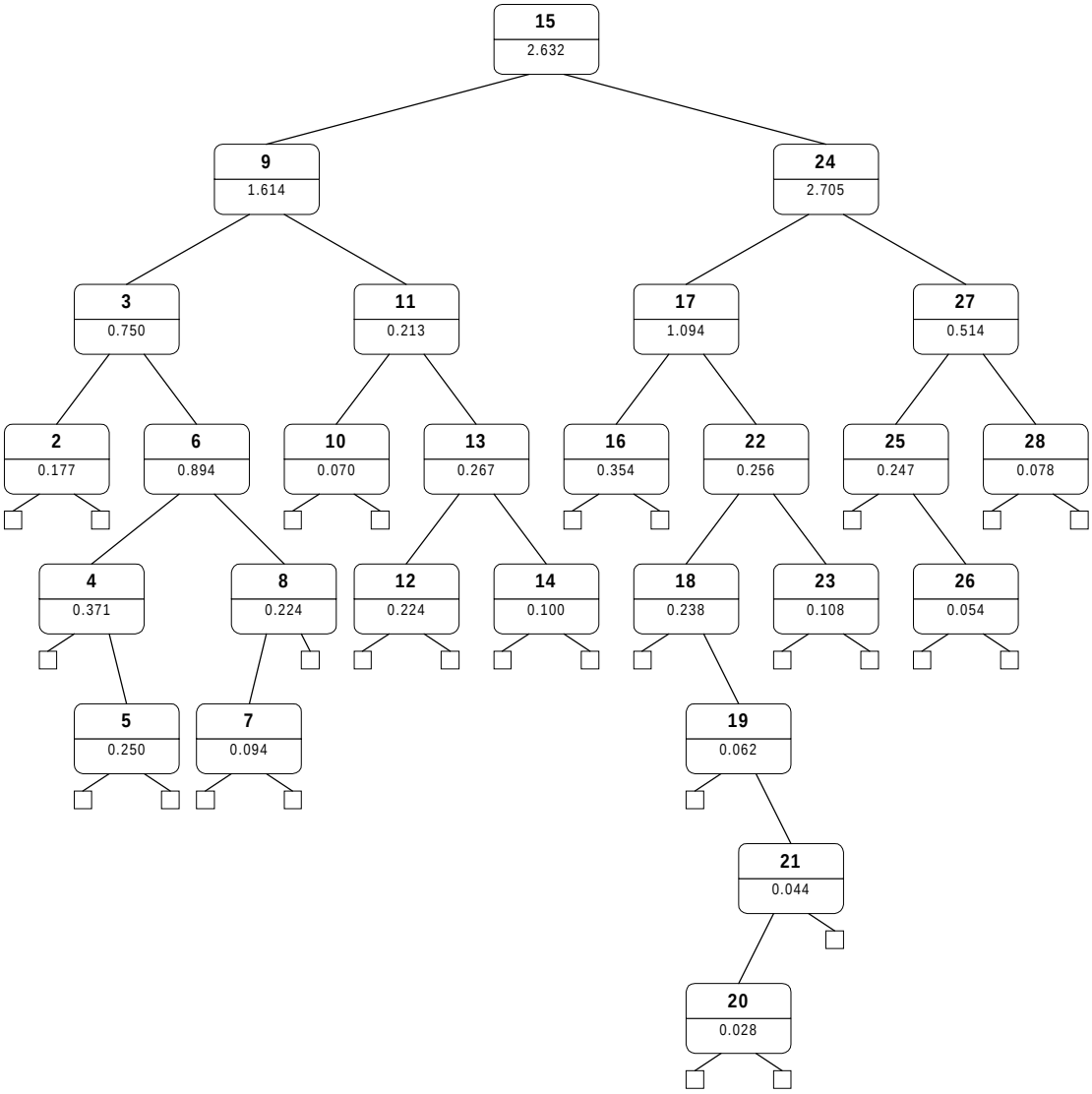


Figura 1.22 - Árvore binária formada a partir do exemplo da Figura 1.15

Complexidade computacional. Como no caso do *quicksort* [Knut73], a análise da complexidade computacional do algoritmo Douglas-Peucker depende da escolha do vértice mais distante, análogo ao pivô do algoritmo de classificação, o que vai determinar o número de cálculos de distância ponto-reta necessários (analogamente ao número de comparações no caso do *quicksort*). Para eliminar a influência do parâmetro de tolerância na análise do pior caso do algoritmo, é necessário considerar que todos os vértices serão selecionados, ou seja, $n' = n$, o que equivale a fazer a tolerância igual a zero.

Assim, a melhor situação para o particionamento ocorre quando o vértice mais distante é o vértice central do intervalo[8]. Neste caso, é possível formular a seguinte equação de recorrência:

[8] Esta situação pode ocorrer, por exemplo, quando os vértices da poligonal estão dispostos ao longo de um semicírculo [Nick88].

$$T(n) = (n-2) + T\left(\left\lfloor \frac{n}{2} \right\rfloor - 2\right) + T\left(\left\lceil \frac{n}{2} \right\rceil - 2\right)$$
$$T(2) = 0 \qquad (1.11)$$

Como geralmente $n >> 2$, pode-se simplificar a Equação 1.11 para obter:

$$T(n) = n + 2T\left(\frac{n}{2}\right)$$
$$T(2) = 0 \qquad (1.12)$$

Resolvendo a Equação 1.12, resulta *O*(*n log n*), que indica o melhor desempenho do algoritmo quando todos os vértices são mantidos. No entanto, o pior caso ocorre, de forma também análoga ao *quicksort*, quando a divisão é feita no segundo vértice ou no penúltimo, fazendo com que a recorrência assuma a seguinte forma:

$$T(n) = (n-2) + T(n-1)$$
$$T(2) = 0 \qquad (1.13)$$

A solução da Equação 1.13 é, portanto, a complexidade computacional do algoritmo Douglas-Peucker no pior caso: $O(n^2)$. O pior caso também pode ocorrer quando o objetivo é formar a estrutura de dados exemplificada na Figura 1.22, pois aquela situação equivale a processar toda a poligonal considerando tolerância zero.

O melhor caso ocorre quando todos os vértices da poligonal podem ser simplificados imediatamente. Assim, apenas uma iteração é necessária, calculando as distâncias de todos os vértices intermediários à reta definida pelos extremos, e portanto o algoritmo é *O*(*n*).

O comportamento em situações reais, portanto, depende fortemente do parâmetro de tolerância e da escolha do vértice mais distante em cada passo. Tolerâncias baixas, significando a preservação de uma quantidade maior de vértices, indicam um tempo de processamento maior. Por outro lado, tolerâncias grandes fazem com que o processamento seja resolvido com poucas iterações, e portanto em menos tempo. Uma análise considerando tolerância zero e quebra em um vértice intermediário escolhido aleatoriamente indica que o tempo de execução médio é um fator de $2 \cdot \log(e) \cong 2{,}885$ sobre o caso linear [HeSn92].

A literatura contém propostas para melhorar o desempenho do algoritmo Douglas-Peucker com o uso de técnicas de geometria computacional. Hershberger et al. [HeSn92] observam que, quando a divisão do problema produz um subproblema com tamanho quase igual ao original, o algoritmo Douglas-Peucker precisa recomeçar o processamento do zero, sem armazenar ou levar em conta qualquer conhecimento sobre a geometria da linha que poderia ter sido obtido na primeira iteração. Assim, propuseram uma variação baseada no *path hull*, uma estrutura de dados baseada no

fecho convexo[9] aplicado a uma linha [DGHS88], buscando maior eficiência na etapa de seleção do vértice mais distante.

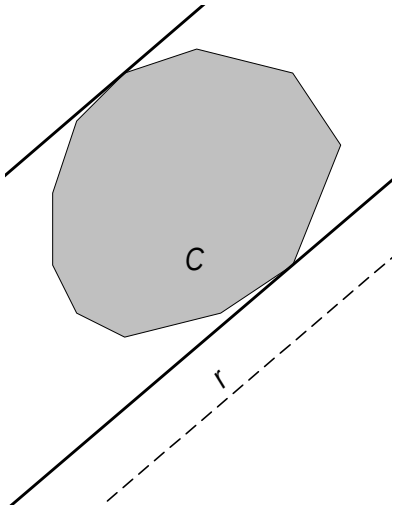


Figura 1.23 - Os pontos do conjunto convexo *C* mais distantes da reta *r* estão em uma das tangentes a *C* paralelas a *r*

A modificação proposta parte do princípio de que o vértice mais distante, utilizado para a divisão, é necessariamente um dos que compõem o fecho convexo dos vértices da poligonal. Esta constatação parte de um lema da geometria computacional que garante que, dados um conjunto convexo C e uma reta r, os pontos de C mais distantes de r estarão em uma das duas tangentes a C paralelas a r (Figura 1.23). Se C é um polígono convexo, então apenas os seus vértices precisam ser considerados para determinar o ponto mais distante de uma dada reta. Considerando a implementação deste conceito, o *path hull* é definido da seguinte maneira:

> ***Definição 1.1*** - Dada uma cadeia de vértices (v_i, ..., v_j) pertencentes a uma poligonal *P*, o *path hull* da cadeia é definido como sendo a tripla [v_m, *CC*(v_i, ..., v_m), *CC*(v_m, ..., v_j)], onde *CC*(v_i, ..., v_j) é o fecho convexo dos vértices entre v_i e v_j. O vértice v_m é chamado de *tag*.

Em seguida, são definidas as operações básicas sobre *path hulls*: criação, divisão e localização do vértice mais distante. A operação de localização do vértice mais distante utiliza uma pesquisa binária sobre os dois fechos convexos do *path hull*, localizando dois pontos extremos em cada. Em seguida, é calculada a distância de cada um destes extremos à reta, e retornado o vértice mais distante em cada fecho. O custo total desta operação é *O*(log *n*) no pior caso. Como esta operação é chamada *O*(*n*) vezes, isto determina o custo total do algoritmo em *O*(*n* log *n*).

A operação seguinte é a criação, que consiste em dividir um conjunto (v_i, ..., v_j) de vértices ao meio, escolhendo o vértice central como *tag*, (*i.e.*, $m = (i + j) / 2$) e criar dois fechos convexos: um entre todos os pontos anteriores ao *tag* (v_i a v_m), e outro com os pontos posteriores (v_m a v_j). Para construir os fechos convexos, os autores indicam a

---

[9] Vide seção 1.1.6

utilização de um algoritmo de Melkman [Melk87], incremental e baseado em pilhas. Estas pilhas conseguem armazenar não apenas o fecho convexo desejado, mas também todos os fechos convexos intermediários, que serão usados a partir das divisões. A operação final, divisão, utiliza estas pilhas para reconstruir o "histórico" do fecho convexo que contém determinado vértice *k*, retrocedendo o fecho previamente calculado para todos os vértices até o momento da inserção de v_k no fecho. Os autores demonstram que a combinação das etapas de criação e divisão custam também $O(n \log n)$ no pior caso.

O algoritmo modificado para incluir o *path hull* tem, desta forma, complexidade computacional $O(n \log n)$ no pior caso, contra $O(n^2)$ no algoritmo original. Os autores chamam a atenção, no entanto, para os fatores constantes, bem mais significativos no caso do *path hull* do que no algoritmo Douglas-Peucker, uma vez que o algoritmo original não trabalha a estrutura geométrica da poligonal. Para desenvolver uma melhor noção comparativa quanto a este aspecto, torna-se necessário implementar ambos os algoritmos no mesmo ambiente computacional e comparar diretamente os tempos de execução na solução de problemas típicos. Os autores do aperfeiçoamento o fizeram, mas somente compararam linhas artificiais, tentando simular os casos extremos de desempenho. No entanto, destacaram a dificuldade de se selecionar um conjunto representativo de linhas para realizar uma comparação adequada.

Um aspecto que caracteriza o algoritmo Douglas-Peucker, e que é utilizado em vários outros algoritmos, é o conceito de *banda de tolerância*, às vezes chamado de *corredor de tolerância*. Visvalingam e Whyatt [ViWh93] observam que, embora a utilização do critério de largura de banda para determinar pontos a eliminar seja bastante razoável, a escolha do vértice mais distante como ponto crítico ou ponto característico da linha é questionável do ponto de vista cartográfico. Dessa forma, contestam também a classificação do Douglas-Peucker como algoritmo global, uma vez que, a partir do momento da separação da linha em duas, a avaliação da linha quanto à continuação do processo de seleção de pontos passa a não mais abranger todos os vértices. Conforme já observado, a preservação do ponto mais distante leva à geração e conservação de "picos" na linha, deteriorando sua aparência em simplificações mais radicais. Alguns autores atribuem este fato à escolha de um valor único para a tolerância, indicando que é mais razoável ter variações neste valor de acordo com a geomorfologia da linha [Horn85][PAF95], mas até o momento nenhuma variação proposta provou resolver integralmente o problema.

Embora exista um conceito geral de que os vértices selecionados pelo algoritmo Douglas-Peucker sejam os "críticos", estudos demonstram que os vértices considerados críticos por humanos nem sempre coincidem com os selecionados pelo algoritmo. Mais grave do que isso é a constatação de que os vértices são selecionados de maneira desbalanceada, isto é, existe a tendência de selecionar vértices demais em áreas mais ricas em detalhes, e vértices de menos nas regiões geometricamente mais simples, causando ao mesmo tempo o efeito já descrito de "picos" e o detalhamento excessivo de áreas pouco significativas no sentido global. Isto é explicado pelo fato do comportamento do algoritmo ser invariante com relação à escala, ou seja, independente da escala o algoritmo continua eliminando *vértices* menos críticos em vez de *feições* menos significativas, e no processo vai deixando para trás vértices "críticos" que terminam por introduzir distorções esteticamente inaceitáveis na linha. Para corrigir esta

distorção, Thapa ([Thap88] *apud* [ViWh90]) observa que alguns dos vértices críticos precisam ser eliminados para que se possa obter linhas generalizadas suaves, uniformes e agradáveis esteticamente. Neste conceito, portanto, existem vértices críticos que não são tão críticos assim.

Outro problema amplamente reportado na literatura [ViWh90] diz respeito à variação que se pode obter no resultado final quando se varia a linha âncora-flutuante inicial. É o caso da aplicação do algoritmo Douglas-Peucker a poligonais fechadas: dependendo do ponto de partida, ou da estratégia de particionamento da poligonal fechada em duas ou mais poligonais abertas, um resultado diferente será obtido.

Mais grave ainda é a possibilidade, já também amplamente documentada na literatura [ViWh90], de que o algoritmo Douglas-Peucker produza, em situações de generalização mais radical, modificações na topologia da linha (como por exemplo auto-interseções), ou modificações na sua situação com relação a linhas vizinhas (como interseções entre curvas de nível simplificadas). Trata-se de um comportamento francamente indesejável, que precisa ser verificado em implementações mais robustas do algoritmo, com o uso de rotinas específicas para detectar este tipo de problema.

1.1.4.8 Zhao-Saalfeld [ZhSa97]

Este algoritmo utiliza uma técnica denominada *sleeve-fitting*, para realizar a simplificação de poligonais em tempo linear [ZhSa97]. O processo é baseado em uma medida angular variável para verificar o atendimento ao critério de tolerância, que apesar disso é expresso em termos de máxima distância perpendicular.

A verificação da tolerância perpendicular através de medidas angulares é feita utilizando alguns conceitos geométricos. Inicialmente, é definida a medida $\alpha(p_1, p_2)$ como sendo o ângulo do vetor p_1p_2 com a horizontal (eixo X), calculado em sentido anti-horário. Observe-se que o sentido é importante, e portanto $p_1p_2 \neq p_2p_1$. A partir de α, é definido o *setor limite*, da seguinte forma (Figura 1.24):

> ***Definição 1.2*** - O setor limite do ponto *p* e ângulos α_1 e α_2 é o conjunto dos pontos do plano cujo ângulo a está entre α_1 e α_2. Ou seja, $A(p,\alpha_1,\alpha_2) = \{q \in R^2 | \alpha_1 \leq \alpha(p,q) \leq \alpha_2\}$.

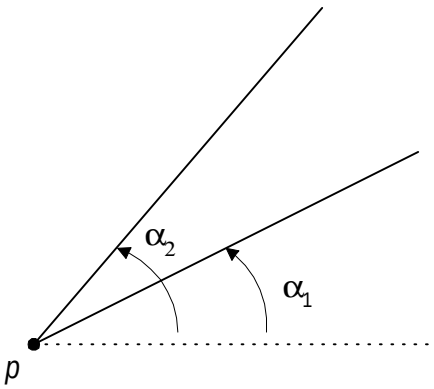


Figura 1.24 - Setor limite

Pode-se sempre assumir que, no setor limite, o ângulo inicial α_1 é menor que o ângulo final α_2. Para garantir isso, basta somar 360^o a α_2 quando este for menor que α_1. Assim, é possível definir a operação de interseção de setores limite baseados no mesmo ponto, da seguinte forma (Figura 1.25):

Definição 1.3 - A interseção de dois setores limite $A(p,\alpha_1,\alpha_2)$ e $A(p,\alpha_3,\alpha_4)$ é um terceiro setor $A(p,\alpha',\alpha'') = A(p,\alpha_1,\alpha_2) \cap A(p,\alpha_3,\alpha_4)$, onde $\alpha' = max(\alpha_1,\alpha_3)$ e $\alpha'' = max(\alpha_2,\alpha_4)$. Se $\alpha' > \alpha''$, então $A(p,\alpha',\alpha'') = \varnothing$.

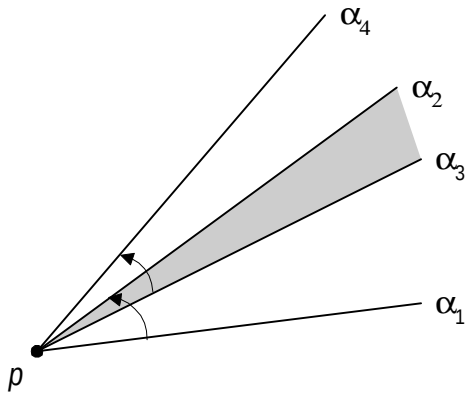


Figura 1.25 - Interseção de setores limite

No caso da distância perpendicular, o interesse dos algoritmos de simplificação está em calcular a distância de um vértice v_k à reta definida por dois vértices da poligonal, v_i e v_j, sendo $i< k < j$. Testar esta distância contra uma tolerância ε equivale, portanto, a verificar se v_k pertence ao setor limite $A(p, \alpha_1, \alpha_2)$ sendo $\alpha_1 = \alpha(v_i, v_j) - \delta$, $\alpha_2 = \alpha(v_i, v_j) + \delta$, e δ = arco seno(ε / $|v_iv_j|$) (Figura 1.26). Isto ocorre quando $\alpha_1 \le \alpha(v_i,v_k) \le \alpha_2$. Assim, é possível testar se um vértice atende ou não ao critério de tolerância apenas calculando o ângulo $\alpha(v_i, v_k)$ em cada passo, uma vez que todos os demais parâmetros (α_1, α_2, δ) são constantes locais.

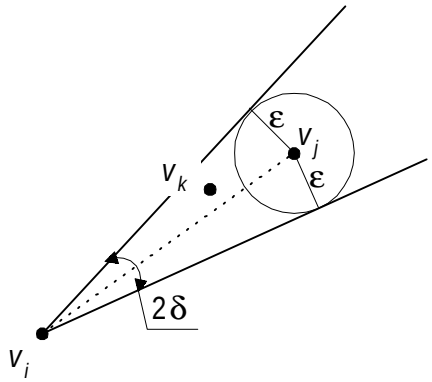


Figura 1.26 - Equivalência entre setor limite e tolerância perpendicular

O algoritmo Zhao-Saalfeld é iniciado calculando o setor limite entre o primeiro e o terceiro vértices, de acordo com o processo descrito acima. O segundo vértice é testado contra este setor limite. Caso esteja fora dele, o segundo vértice é mantido e o processamento recomeça a partir dele. Caso contrário, o segundo vértice será

descartado, e nova avaliação do setor limite é feito entre o primeiro e o quarto vértices. É calculada a interseção entre este setor limite e o setor limite anterior. O terceiro vértice é testado contra a interseção dos setores, caso o resultado não seja o conjunto vazio, e aplicam-se as regras de aceitação e rejeição descritas para o segundo vértice. Se a interseção dos setores limite for anulada, o vértice que está sendo testado (no caso, o terceiro) é mantido, e o processamento recomeça a partir dele.

Note-se que o algoritmo prossegue de forma incremental, e é possível verificar a exclusão de grandes seqüências de vértices sem recorrer a verificações de trás para diante, como no caso do algoritmo de Lang, ou recursões, como no caso do Douglas-Peucker. No entanto, o método é altamente dependente do parâmetro de tolerância, produzindo resultados diferentes de acordo com a variação deste parâmetro. Assim, o algoritmo Zhao-Saalfeld, embora seja bastante eficiente do ponto de vista computacional, não produz uma hierarquia dos vértices de acordo com sua importância na poligonal, e portanto tem sua utilização para generalização dinâmica dificultada. Uma variação, proposta no artigo, permite inclusive a determinação de novos vértices, totalmente diferentes dos originais, formando uma nova poligonal que cabe dentro do mesmo *sleeve* que contém a poligonal original.

No entanto, dada sua natureza incremental e seu comportamento linear, este algoritmo pode ser uma excelente opção para aplicativos de digitalização de linhas, onde trabalharia para eliminar vértices desnecessários à medida em que o trabalho prossegue. Uma demonstração desta possibilidade foi implementada pelos autores em Java, e colocada à disposição no URL `http://ra.cfm.ohio-state.edu/grad/zhao/algorithms/linesimp.html`.

Complexidade computacional. Verifica-se que o processo de reavaliação do setor limite a cada passo confere a este algoritmo um comportamento claramente linear. O único laço presente é, conforme descrito, aquele em que o setor limite específico para o vértice corrente é calculado, e é obtida sua interseção com o resultado da interseção do setor corrente inicial com todos os subseqüentes.

1.1.4.9 Visvalingam-Whyatt [ViWh93]

Este algoritmo propõe uma inversão da lógica utilizada pela maioria dos demais vistos até agora, que tratam de *selecionar* os vértices da poligonal que são necessários para atingir o critério de proximidade. Ou seja, os vértices considerados críticos para a manutenção das características da linha são selecionados e mantidos. No algoritmo de Visvalingam-Whyatt, ao contrário, os pontos *menos* significativos são progressivamente eliminados [ViWh93].

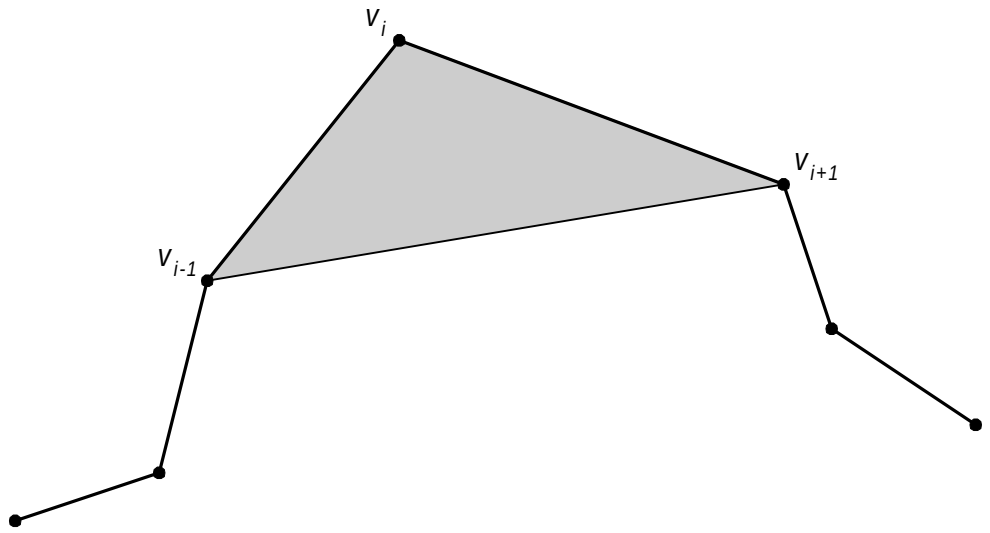


Figura 1.27 - Área efetiva do vértice v_i

Também no critério de proximidade este algoritmo se distingue dos demais, uma vez que utiliza o conceito de *área efetiva*, em vez de distâncias ou ângulos. A área efetiva correspondente a um vértice v_i é a área do triângulo formado pelos vértices v_{i-1}, v_i e v_{i+1} (Figura 1.27). A cada passo, o vértice com menor área efetiva é eliminado, e a área efetiva dos dois vértices adjacentes a ele é recalculada, desconsiderando o vértice eliminado (Programa 1.16).

```
Procedimento Visvalingam-Whyatt(linha, numvert, tol_area)

início
     para i = 1 até numvert - 2 faça
       calcular a área efetiva do vértice i;

     repita
         min = vértice com menor área efetiva correspondente;
         se area_efetiva(min) < tol_area
             eliminar o vértice min;
         recalcular as áreas efetivas dos vizinhos do ponto eliminado;
     até que (area_efetiva(min) >= tol_area) ou
         (todos os vértices intermediários foram eliminados);
fim.
```

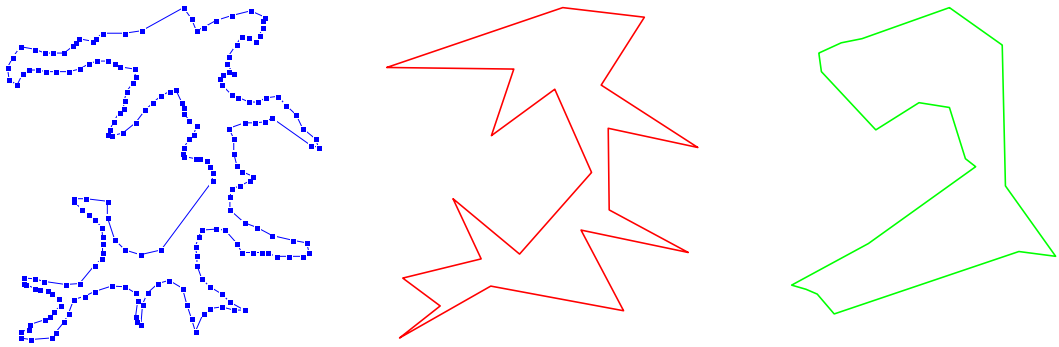
Programa 1.16 - Algoritmo Visvalingam-Whyatt

O critério de eliminação baseado na área efetiva se inspira precisamente nas medidas propostas por McMaster [McMa86] (vide seção 1.1.4.1); no entanto, a medida proposta é global (linha simplificada *versus* linha original), enquanto no algoritmo a avaliação é feita a cada passo, ou seja, entre a versão anterior da linha e a versão atual. É possível armazenar uma lista contendo os pontos eliminados em ordem, juntamente com a área efetiva correspondente, para que seja possível hierarquizar o resultado e repetir a simplificação dinamicamente. No entanto, Visvalingam e Whyatt [ViWh90] alertam para o problema de que não é possível obter um ranqueamento dos vértices simplificados que seja universalmente aceitável, e demonstram este fato utilizando comparações entre linhas simplificadas manualmente e usando o algoritmo Douglas-Peucker. Como o Douglas-Peucker é hierárquico e trabalha recursivamente, em ocasionalmente são selecionados vértices em situações particulares (por exemplo, no

meio de curvas abertas) em estágios preliminares do algoritmo, impedindo que outros vértices, perceptivelmente mais críticos (por exemplo, nas extremidades dessas curvas abertas) sejam selecionados, distorcendo portanto o resultado final.

Um problema o algoritmo Visvalingam-Whyatt, destacado pelos próprios autores, é a escolha do valor de parada: a partir de que área efetiva o procedimento deve ser interrompido? Uma correlação direta da área efetiva com a distância perpendicular, como utilizada no Douglas-Peucker, não pode ser feita, pois a distância entre os vértices extremos pode variar. No entanto, os experimentos realizados pelos autores indicam que o algoritmo oferece oportunidades para simplificação mínima (apenas filtragem de vértices desnecessários), utilizando tolerâncias pequenas, e também para generalização. Isto provavelmente deriva do fato de se utilizar uma medida local – a área efetiva – considerada globalmente a cada passo (eliminando o ponto com menor área efetiva entre todos os remanescentes na linha).

Geometricamente, o algoritmo se comporta de maneira diferente do algoritmo Douglas-Peucker, uma vez que apresenta uma tendência a "cortar cantos", e a eliminar progressivamente as características inerentes ao tamanho da feição (Figura 1.28). O resultado visual é mais interessante em simplificações mais radicais, pois produz uma "caricatura" livre das distorções grosseiras que caracterizam os resultados da aplicação do algoritmo Douglas-Peucker com tolerâncias grandes (na Figura 1.28b foi utilizada tolerância de 100 metros, equivalente a 2mm em escala 1:50.000), como a geração de "picos" (Figura 1.28b). Por outro lado, a eliminação de uma quantidade menor de vértices produz uma simplificação mínima razoável, semelhante à produzida pelo algoritmo Douglas-Peucker.



(a) **(b)** **(c)**

Figura 1.28 - Lago digitalizado com 204 vértices (a), e simplificado pelos algoritmos de Douglas-Peucker (b) e Visvalingam-Whyatt (c), mantendo 10% dos vértices originais.

A escala de digitalização do lago da Figura 1.28 foi 1:50.000. A Figura 1.29 compara as três representações da Figura 1.28, desta vez plotadas em escalas aproximadas de 1:100.000 e 1:200.000. Observe-se como o detalhamento obtido na digitalização original torna a poligonal muito complexa, com detalhes praticamente irreconhecíveis, nas duas escalas menores (Figura 1.29a), produzindo um aspecto excessivamente poluído. A simplificação por Douglas-Peucker (Figura 1.29b) efetivamente diminui a quantidade de detalhes, mas apresenta vários "picos" com aspecto pouco natural. A

simplificação por Visvalingam-Whyatt (Figura 1.29c) consegue eliminar as feições menos significativas do lago, como os diversos pequenos estuários que ocorrem ao seu redor.

(a) **(b)** **(c)** **(d)**

Figura 1.29 - Lago da Figura 1.28, original (a) e simplificações por Douglas-Peucker (b) e Visvalingam-Whyatt (c e d), plotado em escalas 1:100.000 e 1:200.000

Um aspecto ainda melhor poderia ser alcançado caso se permitisse a utilização do Visvalingam-Whyatt com mais vértices do que os retidos pelo Douglas-Peucker. Na Figura 1.29d, por exemplo, a representação utilizou o dobro dos vértices que foram mantidos na Figura 1.29b e Figura 1.29c, obtendo uma aparência mais natural, mas ainda razoavelmente simples.

Complexidade computacional. A implementação deste algoritmo necessita de alguma estrutura de dados que facilite a seleção, a cada passo, do vértice cuja área efetiva seja mínima. A escolha natural é uma fila de prioridades, sob a forma de um *heap* binário. Nessa estrutura de dados, a raiz contém sempre o menor elemento, e elementos de um determinado nível são sempre menores que os elementos do nível hierarquicamente inferior. Assim, a inserção inicial (criação do *heap*) é feita em tempo $O(n \log n)$, enquanto a retirada é feita em tempo constante [Zivi96] [Sedg90].

Além disso, esta estrutura de dados precisa ser modificada em cada iteração do algoritmo, uma vez que a eliminação de um vértice causa o recálculo das áreas efetivas dos dois vizinhos imediatos, o que por sua vez demanda um rearranjo no *heap* para preservar sua propriedade. No caso do algoritmo, pode ocorrer que uma área efetiva já presente no *heap* seja modificada para menos ou para mais. No primeiro caso, o item correspondente à área modificada precisa subir no *heap*, enquanto no segundo caso o item precisa descer. Qualquer rearranjo desta natureza em *heaps* é feito em tempo $O(\log n)$. Como o rearranjo é feito a cada retirada de elementos, o pior caso é a execução deste passo n vezes, e portanto temos novamente tempo $O(n \log n)$, que é a complexidade final do algoritmo.

1.1.4.10 Chan-Chin [ChCh96]

Este algoritmo propõe uma formulação diferente para o problema de simplificação, baseada na constatação de que a maioria dos algoritmos usualmente encontrados na literatura não oferece qualquer garantia de otimalidade na solução [ChCh96]. A idéia é obter, por meio de técnicas de otimização bastante conhecidas, a solução para um dos seguintes problemas:

- **mínimo número de segmentos:** dada uma poligonal P e uma tolerância ε, construir uma nova poligonal P' com distância perpendicular inferior a ε, de modo que esta poligonal tenha o menor número possível de segmentos;

- **mínimo erro:** dada uma poligonal P e um número máximo de segmentos (m), construir uma poligonal aproximada P' com m ou menos segmentos de modo que o erro total seja mínimo.

O problema de mínimo erro não tem aplicação imediata na área de interesse deste trabalho, uma vez que desejamos construir poligonais mais compactas dado um erro máximo tolerável. Já o problema de número mínimo de segmentos se assemelha bastante em termos de formulação a alguns dos algoritmos já analisados, introduzindo o requisito de garantia de otimalidade, ou seja, de que o resultado tenha um número mínimo de segmentos e portanto um número mínimo de vértices.

Para a solução do problema do mínimo número de segmentos, o primeiro passo é construir um grafo $G = (V, E)$, onde cada vértice $v \in V$ representa um vértice v_i da poligonal original P. O conjunto de arestas E é formado por todos os pares (r, s) tais que o erro correspondente ao segmento $v_r v_s$ é inferior a ε, ou seja, $E = \{(r, s) \mid r < s$ e $erro(r, s) \leq \varepsilon\}$. A função *erro* é exatamente a que determina a distância perpendicular, conforme utilizada em diversos outros algoritmos, como o Douglas-Peucker. O grafo G é chamado de ε-grafo de P.

A solução do problema consiste em encontrar o menor caminho em G de v_0 até v_{n-1}, considerando custo unitário para cada aresta de E. Os vértices encontrados nesse caminho mínimo serão exatamente os vértices da poligonal aproximada P', e o comprimento do caminho mínimo informa quantos segmentos existirão em P'. Esta fase pode ser executada em tempo $O(n \log n)$, que é o melhor tempo para o algoritmo clássico de Dijkstra [Baas88] [Tarj83].

O principal problema consiste em montar o grafo G, checando cada par de vértices para verificar o atendimento à condição de distância máxima. Isto pode ser feito na base da força bruta, combinando todos os vértices da poligonal entre si, mas levaria tempo $O(n^3)$, uma vez que existem $O(n^2)$ pares de vértices e para cada par seria necessário verificar o erro avaliando a distância dos vértices intermediários, o que pode consumir tempo $O(n)$. Outros autores apresentaram propostas semelhantes, obtendo tempo $O(n^2 \log n)$ [Melk88] [ImIr86], mas Chan e Chin demonstraram a viabilidade de obter o mesmo resultado em tempo $O(n^2)$. De qualquer maneira, este tempo é dominante sobre o tempo de processamento do caminho mínimo, e superior aos apresentados pelos algoritmos anteriores, que não garantem resultados ótimos.

Um outro problema a considerar é determinar se a escolha de vértices que minimizam o número de segmentos da poligonal resultante é capaz de produzir resultados aceitáveis do ponto de vista da estética e funcionalidade cartográficas. Além disso, existe o problema do tempo de processamento e o fato de o algoritmo não produzir uma hierarquia de vértices na saída, sendo portanto de aplicação pouco indicada para generalização dinâmica. No entanto, como os resultados produzidos são garantidamente ótimos quanto ao número de vértices e segmentos, torna-se possível utilizar o resultado deste algoritmo para verificar e comparar a eficiência dos demais quanto à compactação da poligonal.

1.1.4.11 Li-Openshaw [LiOp92]

Li e Openshaw [LiOp92] formularam uma série de algoritmos para simplificação de linhas a partir de um chamado "princípio natural" da generalização. Este princípio parte da constatação de que existem limites para a capacidade humana de perceber detalhes, e também limites quanto à resolução dos dispositivos utilizados para visualização, como os monitores de vídeo dos computadores atuais. Por exemplo, considere-se se um determinado objeto poligonal, desenhado na tela do computador utilizando sua máxima resolução. À medida em que se diminui a escala de apresentação, o objeto diminui de tamanho na tela, utilizando uma quantidade menor de pixels para sua representação. Eventualmente, a escala diminui tanto que o objeto se transforma em um único pixel, e depois menos do que isso, sendo então impossível representá-lo na tela em suas dimensões aproximadas.

Desta forma, a concepção dos algoritmos parte da dedução do tamanho do menor objeto visível (*smallest visible object*, ou SVO). Naturalmente, o tamanho do SVO é expresso nas unidades de medida da tela ou do mapa, e portanto pode ser traduzido em medidas reais através da aplicação do fator de escala. Li e Openshaw utilizam um resultado anterior, obtido por Müller [Mull87], que indica o valor de 0,4mm como sendo o mínimo necessário para assegurar a separabilidade visual dos objetos, com base na espessura das linhas plotadas e na resolução do olho humano. Qualquer detalhe de tamanho menor é desnecessário: virtualmente toda a informação adicional que exista é perdida, pois os humanos não são capazes de percebê-la. Outro estudo do mesmo autor [Mull90a] usa o mesmo conceito na detecção de conflitos entre elementos generalizados.

Li e Openshaw propuseram três algoritmos diferentes: vetorial, *raster*, e um terceiro que combina ambos. No primeiro, é gerado um círculo cujo diâmetro é o dobro tamanho do SVO sobre o primeiro vértice. Todo detalhamento interior ao círculo é desprezado, e é substituído por um "centróide", que é o ponto médio do segmento unindo o vértice inicial (centro do círculo) e o ponto onde o círculo intercepta a poligonal. Nesse último ponto, o algoritmo é reiniciado, repetindo a mesma operação até que o vértice final seja inserido em um círculo. O resultado é interessante, pois força a eliminação de detalhes de tamanho inferior ao do SVO, produzindo representações bastante suavizadas e de aspecto real. No entanto, este algoritmo não seleciona vértices da poligonal original, e portanto é não-hierárquico. Outro problema é que não é forçada a utilização do primeiro e do último vértices, o que pode gerar inconsistências topológicas na visualização.

Na versão *raster* do algoritmo é utilizado, em lugar do círculo, um pixel de tamanho equivalente ao do SVO. O processamento é essencialmente o mesmo da versão vetorial, com a diferença de estarem sendo escolhidos pixels específicos, dentro de uma matriz, que ficarão "ligados" para representar a poligonal.

A versão *raster-vector* combina os pontos positivos das duas alternativas anteriores. É utilizado um quadrado, como na versão *raster*, mas que pode estar posicionado arbitrariamente, como o centróide da versão vetorial. Os resultados da aplicação deste algoritmo apresentam boa consistência geométrica, especialmente quando comparados com resultados do algoritmo Douglas-Peucker em simplificações radicais [LiOp92]. No entanto, os autores apontam problemas na simplificação de corredores estreitos, que podem ser fundidos em uma só linha, gerando uma aparência indesejável. Outro

problema apontado corresponde aos critérios de escolha do tamanho do SVO para geração de representações em outras escalas. Testes e experiências realizados pelos autores indicam como adequado valores da ordem de 0,5 a 0,6 mm.

1.1.4.12 Comparação entre os algoritmos

Como foi visto nesta seção, existem muitos algoritmos para simplificação de poligonais, com boa eficiência computacional e comportamento geométrico aceitável dentro de limites. Dentre os algoritmos apresentados, o Douglas-Peucker se destaca por ser o mais utilizado nos sistemas comerciais, e por viabilizar a construção de uma árvore binária para estruturar o resultado do processamento. É também um algoritmo que apresenta bons resultados na simplificação com tolerâncias baixas, com a manutenção de uma parcela significativa dos vértices. No entanto, seu comportamento geométrico deixa a desejar em situações de simplificação mais radical, com mudanças de escala relativamente grandes. O algoritmo Visvalingam-Whyatt parece oferecer uma solução mais interessante para a simplificação radical, mantendo a possibilidade de estruturar os resultados para armazenamento. No entanto, a fixação do parâmetro de tolerância pode ser problemática. O algoritmo Zhao-Saalfeld é uma alternativa interessante e rápida para a filtragem de vértices desnecessários ao longo do processo de digitalização. Seu funcionamento incremental, no entanto, dificulta sua utilização para generalização dinâmica, pois não existe maneira óbvia de estruturar e armazenar seus resultados.

1.1.5 Geometria de polígonos

Uma parcela importante do trabalho de geometria computacional é realizada sobre polígonos. Estes constituem uma maneira conveniente de representar, em um computador, entidades do mundo real cuja abstração transmite a noção de área. Estes tipos de objetos são muito comuns em SIG, sendo muitas vezes denominados “objetos de área”, e são usados para representar graficamente entidades bidimensionais, tais como o contorno de edificações, propriedades, regiões de uso do solo e, genericamente, todo tipo de divisão territorial, tais como estados, municípios, bairros e setores censitários. A maioria dos polígonos encontrados em bancos de dados geográficos são muito simples, mas podem existir objetos poligonais muito complicados. Por exemplo, é muito provável que o formato de uma edificação seja muito mais simples que a fronteira entre dois municípios, uma vez que o primeiro é criado pelo homem, enquanto o segundo é tipicamente definido por elementos naturais, tais como rios ou divisores de águas.

Dado o uso intensivo de polígonos em SIG, e a natureza das aplicações usuais, os algoritmos empregados para trabalhar com polígonos precisam ser escolhidos cuidadosamente. Neste sentido, a eficiência pode ser uma decorrência da divisão do polígono em partes mais simples, como triângulos. Nesta seção, serão apresentados algoritmos que exigem a divisão do polígono em triângulos, como o cálculo de área e centro de gravidade, além do algoritmo para triangulação propriamente dito. Além disso, serão apresentados elementos de aritmética de polígonos, incluindo união, interseção e criação de *buffers*.

1.1.5.1 Propriedades básicas e triangulação

Definições. Um *polígono* é a região do plano limitada por uma coleção finita de segmentos, formando uma *curva fechada simples*. Um *segmento* é o subconjunto fechado dos pontos de uma reta compreendidos entre dois *pontos extremos*.

Mais formalmente, um polígono pode ser definido como se segue:

> ***Definição 1.4*** - Sejam $v_0, v_1, \ldots, v_{n-1}$ n pontos do plano. Sejam $a_0 = \overline{v_0 v_1}, a_1 = \overline{v_1 v_2}, \ldots, a_{n-1} = \overline{v_{n-1} v_0}$ n segmentos, conectando os pontos. Estes segmentos limitam um polígono P se, e somente se, (1) a interseção de segmentos adjacentes é unicamente o ponto extremo compartilhado por eles (ou seja, $a_i \cap a_{i+1} = v_{i+1}$), e (2) segmentos não adjacentes não se interceptam (ou seja, $a_i \cap a_j = \varnothing$ para todo *i, j* tais que $j \neq i+1$).

Os pontos v_i e os segmentos a_i são chamados respectivamente os *vértices* e as *arestas* do polígono. As arestas formam uma *curva* porque são conectados seqüencialmente, pelos pontos extremos. A curva é *fechada* porque o último segmento conecta o último vértice ao primeiro. A curva fechada é denominada *simples* porque não existem interseções de arestas não adjacentes. Se a condição (2) não for observada, ainda assim o objeto é chamado, em muitas situações, de polígono, mas será um polígono *não-simples* (Figura 1.30). Em geral, todos os polígonos mencionados de agora em diante serão considerados simples, a menos que explicitamente se afirme o contrário. A definição acima poderia ser baseada na definição de linha poligonal, uma vez que o polígono é exatamente uma poligonal fechada. O fato de ser fechada, no entanto, acrescenta propriedades importantes, como veremos a seguir.

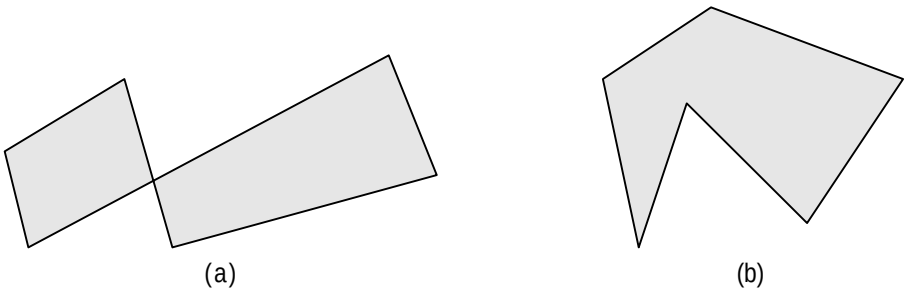


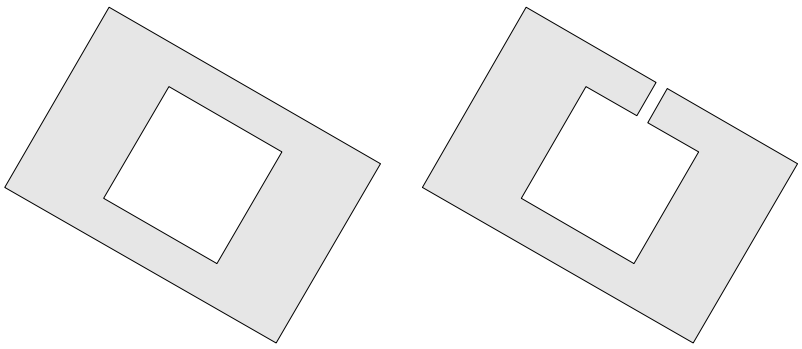
Figura 1.30 - Polígonos não-simples (a) e simples (b)

Outro conceito importante envolvendo polígonos é o de *convexidade*. Um conjunto *S* de pontos é dito *convexo* se, para quaisquer dois pontos p_1 e p_2 em *S* o segmento p_1p_2 está inteiramente contido em *S*. Um polígono simples é então dito *convexo* se seu interior forma um conjunto convexo.

Note-se que o polígono, sendo a região do plano limitada pelas arestas, contém todos os pontos internos. Em muitas aplicações e algoritmos, no entanto, existe a necessidade de distinguir entre a *fronteira* e a região limitada pelo polígono. Assim, será usada aqui a notação ∂P para representar a fronteira, enquanto *P* representará a região. Isto significa que $\partial P \subseteq P$. O símbolo de derivada parcial é usado porque, no sentido topológico, a fronteira é a derivada parcial da região. A partir desta definição, concluímos que o polígono, entendido como sendo a região, divide o plano em duas partes: o *interior*

(limitado) e o *exterior* (ilimitado). Este conceito é fundamental para aplicações de SIG envolvendo polígonos.

Observe-se também que a definição acima não considera a possibilidade da existência de “ilhas” ou “buracos” no polígono, uma vez que o polígono deve conter todos os pontos do plano no interior da fronteira. Em algumas definições de polígonos encontradas na literatura, tal possibilidade é explicitamente descartada pelo uso de uma regra que afirma que nenhum subconjunto das arestas do polígono compartilha a propriedade do polígono, ou seja, nenhum subconjunto de arestas forma uma curva fechada simples. Tais objetos, no entanto, freqüentemente aparecem em problemas práticos, como o contorno de edificação apresentado na Figura 1.31. O edifício tem um “buraco” em seu contorno, que funciona como coluna de ventilação. A área ocupada pelo edifício pode ser aproximada por um polígono no qual o que era um buraco passa a ser conectado com o exterior. Naturalmente, a largura do corredor entre o buraco e o exterior determina a qualidade da aproximação, tanto visualmente (esteticamente) quanto numericamente. Esta técnica é amplamente utilizada em SIG, para preservar a correção de mapas temáticos e a possibilidade de gerar cálculos de área mais aproximados.



(a) polígono com buraco (b) buraco aproximado

Figura 1.31 - Aproximação de buraco em polígono

Em geometria computacional a seqüência dos vértices de um polígono é dada em sentido anti-horário, por convenção. Este não é o caso da maioria dos produtos de SIG, que não exigem qualquer ordenação especial dos vértices de um polígono para aceitá-lo no banco de dados geográfico. Alguns nem mesmo verificam se o polígono é ou não simples, antes de aceitar sua inclusão. Isto pode levar a problemas, como veremos posteriormente.

Alguns problemas e aplicações de geometria computacional usam a noção de polígonos *estrelados* [FiCa91]. São polígonos para os quais existe um ponto *z*, não-externo, tal que para todos os pontos *p* contidos no polígono o segmento *zp* está inteiramente contido no polígono. O conjunto *Z* de pontos que compartilham esta propriedade é denominado *núcleo* do polígono. Naturalmente, todos os polígonos convexos são em formato de estrela, e o seu núcleo coincide com o próprio polígono. Outros exemplos de polígonos estrelados estão apresentados na Figura 1.32.

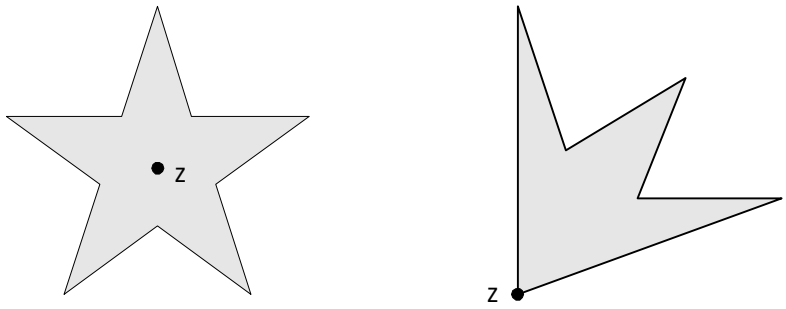


Figura 1.32 - Polígonos estrelados e pontos pertencentes ao núcleo (*z*)

O Problema da Galeria de Arte. O Problema da Galeria de Arte é um clássico da geometria cuja solução conduz naturalmente ao problema de triangulação de polígonos, importante em geoprocessamento. Neste problema, a planta poligonal de uma galeria de arte deve ser *coberta* (vigiada) por um determinado número de guardas estacionários. Cada guarda pode enxergar tudo ao seu redor mas, naturalmente, não consegue enxergar através das paredes (arestas do polígono). Isso significa que cada guarda tem uma faixa de 2π radianos de *visibilidade*. O problema consiste em determinar quantos guardas são necessários, no mínimo, para vigiar a galeria.

> ***Definição 1.5*** - Um ponto *B* é *visível* por um ponto *A* em um polígono *P* se, e somente se, o segmento fechado *AB* não contém nenhum ponto exterior a *P* ($AB \subseteq P$). *B* é dito *claramente visível* por *A* se *AB* não contém nenhum ponto da fronteira de *P* (∂P) que não *A* e *B* ($AB \cap \partial P = \{A, B\}$).

Observe-se que esta definição permite que os vértices estejam no caminho da visibilidade, sem bloquear a visão do guarda. Definições alternativas poderiam excluir esta possibilidade.

> ***Definição 1.6*** - Um conjunto de guardas *cobre* um polígono *P* se todo ponto em *P* é visível por algum guarda.

Portanto, um guarda é um ponto contido no polígono. Assume-se que os guardas não impedem a visão uns dos outros. A Figura 1.33 apresenta um polígono de 6 lados que pode ser coberto por um único guarda. Como veremos adiante, um guarda não é suficiente para cobrir qualquer polígono de 6 lados.

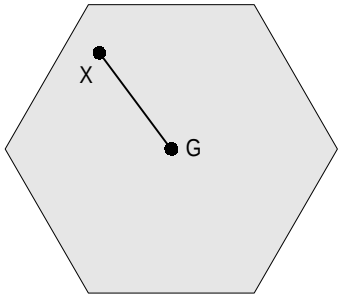


Figura 1.33 - O guarda *G cobre* todo o polígono; o ponto *X* é *visível* a partir de *G*

O Problema da Galeria de Arte procura determinar o máximo, dentre todos os polígonos de *n* lados, do menor número de guardas necessário para cobrir o polígono. Em outras palavras, qual é o menor número de guardas necessário no "pior caso" para um polígono de *n* vértices. Por exemplo, na Figura 1.33 apenas um guarda foi suficiente para cobrir o polígono de 6 lados. A Figura 1.34 mostra dois casos de polígonos de 6 lados que exigem dois guardas. Portanto, a resposta para o Problema da Galeria de Arte é "pelo menos dois guardas". A esta altura, diz-se "pelo menos" porque a prova desta necessidade ainda não foi apresentada.

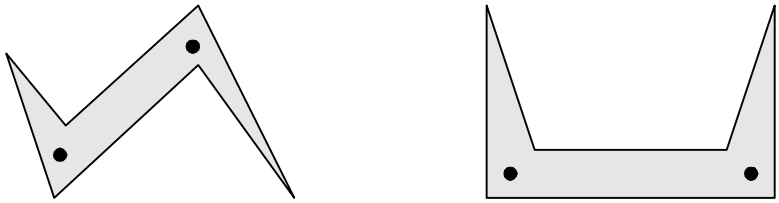


Figura 1.34 - Dois polígonos de seis lados que exigem dois guardas

Para um dado polígono de *n* vértices, existe sempre um número *necessário* (mínimo) de guardas. É necessário mostrar que este número é *suficiente* para *todos* os polígonos de *n* vértices. Formalmente, o problema é determinar os valores de uma função *G(n)*, definida como $G(n) = max_{P_n}\, g(P_n)$, onde P_n é um polígono genérico de *n* vértices, e a função *g(P)* determina o mínimo número de guardas para um polígono específico *P*.

Obviamente, *G*(3) = 1, uma vez que qualquer triângulo é convexo, e portanto pode ser coberto por um único guarda. Pode-se mostrar facilmente que também *G*(4) = 1 e *G*(5) = 1. No entanto, *G*(6) = 2, como se pode perceber pela Figura 1.34, confirmando por experimentação. Isto leva a uma conjetura inicial, de que $G(n) = \lfloor n / 3 \rfloor$. Essa conjetura foi provada por Fisk, em 1978, usando triangulação de polígonos.

A técnica de triangulação de polígonos empregada por Fisk usa a noção de *diagonais*.

> ***Definição 1.7*** - Uma *diagonal* de um polígono P é um segmento cujos pontos extremos são dois de seus vértices v_i e v_j tais que v_i e v_j são *claramente visíveis* um pelo outro.

Portanto, pela definição de "claramente visível", a diagonal não pode interceptar a fronteira do polígono. Duas diagonais são ditas *não-interceptantes* se sua interseção ocorre apenas na fronteira, ou seja, em um dos vértices. Adicionando diagonais não-interceptantes a um polígono, em qualquer ordem, e terminando o processo quando não for mais possível inserir novas diagonais, o polígono é particionado em triângulos. Isto é denominado uma triangulação do polígono, e é sempre possível (vide demonstração em [ORou94]). Em geral, existem muitas maneiras de triangular um polígono, dependendo da seqüência de lançamento das diagonais. Foi provado que qualquer polígono pode ser triangulado usando este processo.

A prova de Fisk inicialmente executa a triangulação de um polígono, formando um grafo no qual os nós são os vértices, e os arcos são as diagonais e os segmentos da fronteira. Fisk mostra que este grafo pode sempre ser colorido com três cores. Uma vez que o processo de coloração seja iniciado pelos vértices de um triângulo arbitrário, os

vértices subseqüentes são sempre forçados a assumir cores, com base no compartilhamento de diagonais e na adjacência de triângulos (a Figura 1.35 contém um exemplo). Já que cada triângulo compartilha pelo menos um de seus lados com outro triângulo (em qualquer polígono com $n \geq 4$), o vértice oposto tem sua coloração forçada pelas escolhas anteriores de cores. Assim, se cada um dos n vértices recebeu uma de três cores, pode-se concluir que pelo menos uma das cores pode ser encontrada em, no máximo, $n / 3$ vértices. Ou seja, $G(n) = \lfloor n / 3 \rfloor$, porque com um guarda em cada um dos vértices correspondentes à cor menos usada é possível cobrir todo o polígono.

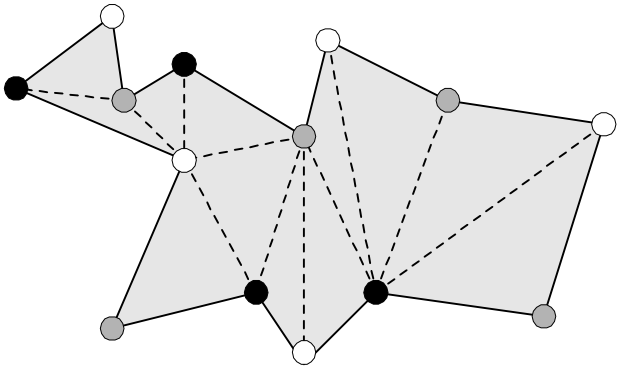


Figura 1.35 - Para este polígono com 14 vértices, 4 guardas são suficientes (nos vértices pretos)

Triangulação de polígonos. Na solução do Problema da Galeria de Arte foi utilizada a triangulação de polígonos, e afirmou-se que a triangulação é sempre possível. Para provar esta afirmação, é necessário estabelecer algumas condições iniciais, relativas à convexidade de vértices e à existência de diagonais.

> ***Definição 1.8*** - Um *vértice estritamente convexo* de um polígono *P* é um vértice no qual, quando se caminha em sentido anti-horário pela fronteira do polígono, faz-se uma curva à esquerda. Analogamente, um *vértice reflexo* é aquele no qual se faz uma curva à direita. Observe-se que, quando se caminha pela fronteira de um polígono em sentido anti-horário, o interior está sempre à esquerda.
>
> ***Lema 1.1*** - *Existe sempre pelo menos um vértice estritamente convexo em um polígono P.*
>
> ***Prova*** - Selecionar, dentre os vértices de *P*, aquele com a menor coordenada *y*. Se existir mais de um vértice com a mesma ordenada, escolher aquele com a maior abscissa. Seja v_i este vértice. Nesta situação, tem-se $y(v_{i-1}) \geq y(v_i)$ e $y(v_{i+1}) > y(v_i)$, e o interior está totalmente acima de v_i. Portanto, é necessário fazer uma curva à esquerda em v_i quando se caminha em sentido anti-horário, de v_{i-1} para v_{i+1} , e portanto v_i é um vértice estritamente convexo (vide Figura 1.36 para uma visualização da prova).

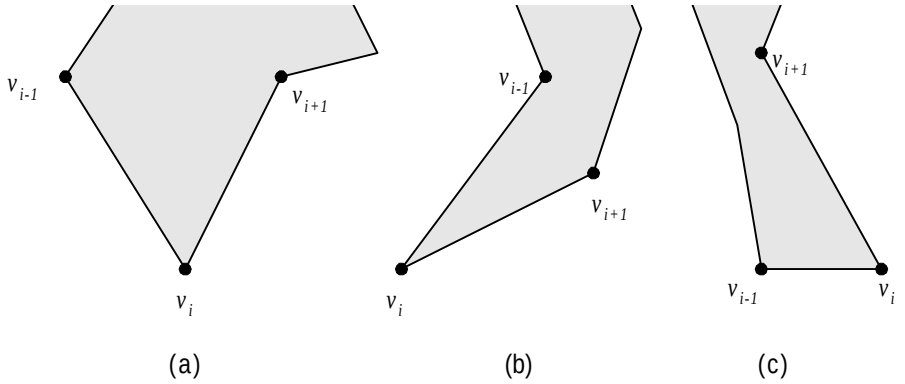


Figura 1.36 - Curva à esquerda no vértice inferior mais à direita, que tem de ser estritamente convexo

Lema 1.2 *- Todo polígono P com mais de 3 vértices tem uma diagonal.*

Prova - Seja v_i um vértice estritamente convexo, cuja existência foi garantida pelo Lema 1.1 . Sejam v_{i-1} e v_{i+1} os vértices imediatamente adjacentes a v_i. Se o segmento $v_{i-1}v_{i+1}$ não for uma diagonal, então ele (i) intercepta ∂P , ou (ii) ele é exterior a P (Figura 1.37). Em ambos os casos, o triângulo $v_{i-1}v_iv_{i+1}$ contém pelo menos um outro vértice de P, uma vez que $n \geq 4$. Um dos vértices incluídos no triângulo $v_{i-1}v_iv_{i+1}$ irá formar uma diagonal com v_i. Este vértice pode ser encontrado pesquisando em uma linha paralela ao segmento $v_{i-1}v_{i+1}$, começando em v_i e deslocando-a em direção a $v_{i-1}v_{i+1}$ até que um vértice x seja encontrado. Já que o triângulo $v_{i-1}v_iv_{i+1}$ foi varrido desde v_i, sem que qualquer outro vértice tivesse sido encontrado, o segmento v_ix não intercepta ∂P num ponto da fronteira diferente de v_i ou x, e portanto constitui uma diagonal.

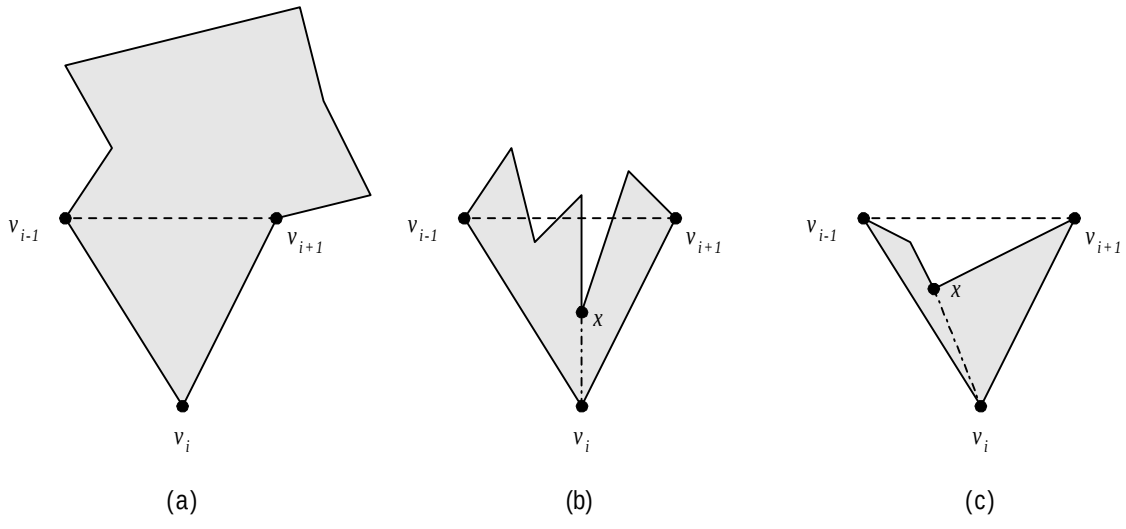


Figura 1.37 - Três situações possíveis no Lema 1.2 **: (a) $v_{i-1}v_{i+1}$ é uma diagonal, (b) $v_{i-1}v_{i+1}$ intercepta a fronteira e (c) $v_{i-1}v_{i+1}$ é exterior a P**

Este lema demonstrou que qualquer polígono com mais de 3 vértices deve ter pelo menos uma diagonal. Observe-se, no entanto, que qualquer diagonal encontrada dividirá

o polígono em duas partes. Se uma parte ficar com apenas três vértices, tem-se obviamente um triângulo. Se ficar com mais de três vértices, formará também um polígono, que também tem pelo menos uma diagonal. Recursivamente, chega-se a uma triangulação completa. O teorema abaixo formaliza esta conclusão.

> ***Teorema 1.2*** - *Todo polígono de n vértices pode ser triangulado*
>
> ***Prova*** - Se $n = 3$, o polígono já é um triângulo. Se $n \geq 4$, o polígono pode ser dividido pela adição de uma diagonal, usando o processo descrito no Lema 1.2 . Uma diagonal, quando encontrada, divide o polígono em dois sem adicionar novos vértices. Cada uma das partes é um polígono, compartilhando uma aresta, que é a diagonal, e cada uma das partes tem menos de n vértices. Prosseguindo, cada parte pode ser dividida recursivamente até que se chegue a uma triangulação completa.

A triangulação de um polígono tem diversas propriedades, que são úteis para as aplicações.

> ***Lema 1.3*** - *Toda triangulação de um polígono P com n vértices usa n - 3 diagonais e gera n - 2 triângulos.*
>
> ***Prova*** - A prova é por indução sobre o número de vértices.
>
> *Base:* Se $n = 3$, então existem $3 - 3 = 0$ diagonais e $3 - 2 = 1$ triângulo.
>
> *Hipótese indutiva:* Se $n \geq 4$, então existem $n - 3$ diagonais e $n - 2$ triângulos.
>
> *Passo indutivo:* Sejam n_1 e n_2 o número de vértices dos polígonos P_1 e P_2, obtidos pela divisão de P por uma diagonal d. Naturalmente, $n = n_1 + n_2 + 2$, uma vez que os pontos extremos da diagonal d são contados duas vezes. Se a hipótese indutiva for verdadeira, então P_1 terá $n_1 - 3$ diagonais e gerará $n_1 - 2$ triângulos, e P_2 terá $n_2 - 3$ diagonais e gerará $n_2 - 2$ triângulos. No total, existirão $(n_1 - 3) + (n_2 - 3) + 1 = n - 3$ diagonais (incluindo d), e $(n_1 - 2) + (n_2 - 2) = n - 2$ triângulos, o que confirma a hipótese indutiva.

Um corolário ao Lema 1.3 é que a soma dos ângulos internos de um polígono com n vértices é $(n - 2)\pi$, correspondendo à soma da contribuição de cada um dos $(n - 2)$ triângulos, cada qual tendo somatório de ângulos internos igual a π.

Em seguida, uma série de resultados que correlacionam triangulações a grafos serão apresentados. Estas propriedades são úteis em diversas situações práticas, como será apresentado até o fim deste capítulo.

> ***Definição 1.9*** - O *dual* da triangulação de um polígono é um grafo, formado por nós que correspondem a cada triângulo, e arcos ligando nós cujos triângulos correspondentes compartilham uma diagonal (Figura 1.38).

No grafo dual, cada nó claramente terá grau máximo igual a 3, uma vez que cada triângulo tem no máximo três lados para compartilhar.

> ***Lema 1.4*** - *O dual de uma triangulação não tem ciclos.*

Prova - vide [ORou94].

Se o dual de uma triangulação não tem ciclos, então trata-se de uma árvore. Como cada nó tem grau máximo de 3, então a árvore é binária, quando se escolhe uma raiz com grau 1 ou 2. Esta correspondência entre triangulação de polígonos e árvores binárias é muito útil na implementação de algoritmos de geometria computacional.

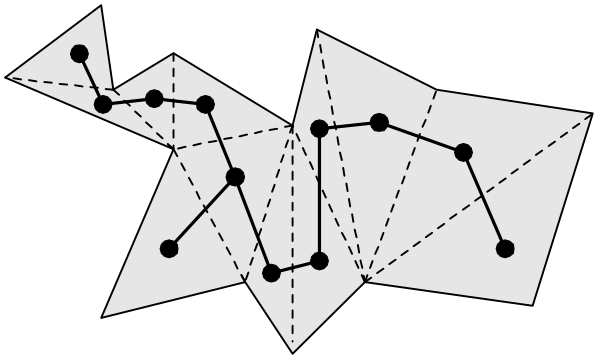


Figura 1.38 - Uma triangulação e sua árvore dual

1.1.5.2 Área de um polígono

A área de um polígono pode ser calculada em tempo linear com relação ao número de vértices, usando um somatório simples, baseado na soma de áreas de triângulos.

O cálculo pode ser feito como se segue. Sejam x_i e y_i as coordenadas do vértice v_i do polígono P, com n vértices. A área do polígono é dada por

$$A(P) = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - y_i x_{i+1}) \qquad (1.14)$$

Observe-se que, na expressão acima, quando se tem $i = n-1$, é necessário ter $x_n = x_0$ e $y_n = y_0$, de acordo com a definição de polígono, caracterizando o seu fechamento.

O sinal da área calculada indica o sentido da seqüência de vértices. A área será negativa se os vértices estiverem em sentido horário, ou positiva se em sentido anti-horário, exatamente como no caso da área do triângulo (Equação 1.1). Como já foi dito, a base do raciocínio para o desenvolvimento do somatório é o mesmo do cálculo da área de um triângulo. O somatório da Equação 1.14 corresponde à soma da área de n triângulos, formados entre um ponto arbitrário (no caso, a origem do sistema de coordenadas) e cada par seqüencial de vértices (v_i, v_{i+1}). A demonstração pode ser encontrada em [FiCa91].

Caso se deseje simplesmente determinar a orientação dos vértices, existe ainda um método mais rápido. Basta determinar o vértice inferior e mais à direita do polígono, e então calcular o produto vetorial das duas arestas que se conectam naquele vértice. Isso funciona porque, como já apresentado anteriormente, este vértice é necessariamente convexo, e portanto o ângulo interno baseado nele é menor que 180^o. Então, a orientação global do polígono pode ser deduzida a partir da orientação do triângulo

formado entre este vértice e seus vizinhos imediatos. Este método é também $O(n)$, mas evita todas as multiplicações e somas necessárias para o cálculo da área.

1.1.5.3 Determinação do centróide de um polígono

Em muitas situações práticas, é necessário determinar, dado um polígono qualquer, seu *centro de gravidade* ou *centro de massa*, mais conhecido em SIG como *centróide*. Em SIG, o centróide é muitas vezes criado e relacionado ao polígono para viabilizar o armazenamento de dados alfanuméricos associados no banco de dados geográfico. É também usado como ponto de lançamento automático de textos gráficos, para identificação de elementos em tela e plotados.

O centróide de um polígono pode ser obtido a partir da sua divisão em triângulos, calculando em seguida a média ponderada dos centros de gravidade dos triângulos usando suas áreas como peso[10]. O centro de gravidade de cada triângulo é simplesmente a média das coordenadas de seus vértices, ou seja, as coordenadas do centro de gravidade de um triângulo ABC seriam:

$$x_G = \frac{x_A + x_B + x_C}{3} \text{ e } y_G = \frac{y_A + y_B + y_C}{3}$$

Embora este processo seja relativamente simples, pressupõe-se a implementação de um algoritmo de triangulação de polígonos. Os centróides dos triângulos são combinados usando um processo de média ponderada pela área. Assim, o centróide de um polígono formado por dois triângulos T_1 e T_2, cujos centróides são, respectivamente, (x_{G1}, y_{G1}) e (x_{G2}, y_{G2}) é o ponto (x_G, y_G), onde

$$x_G = \frac{x_{G1}S(T_1) + x_{G2}S(T_2)}{S(T_1) + S(T_2)}$$
$$y_G = \frac{y_{G1}S(T_1) + y_{G2}S(T_2)}{S(T_1) + S(T_2)}$$

e o centróide do polígono pode ser determinado de maneira incremental, adicionando um triângulo e seu centróide por vez e calculando as coordenadas do centróide do conjunto.

No entanto, existe uma solução mais simples e independente da triangulação, e que leva em conta triângulos com áreas positivas e negativas, como no cálculo da área do polígono. O mesmo processo de média ponderada pela área pode ser usado, considerando todos os triângulos formados entre um ponto fixo, por exemplo (0, 0), e cada par de vértices sucessivos, (v_i, v_{i+1}).

Assim, temos que

---

[10] Este resultado pode ser demonstrado usando as coordenadas baricêntricas, conforme apresentado na seção 1.1.2.1, atribuindo valores iguais a 1/3 para os parâmetros λ_1, λ_2 e λ_3 (Equação 1.3).

$$A(P) = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - y_i x_{i+1})$$

$$x_C = \frac{\sum_{i=0}^{n-1}(x_{i+1} + x_i) \times (x_i y_{i+1} - y_i x_{i+1})}{3A(P)} \qquad (1.15)$$

$$y_C = \frac{\sum_{i=0}^{n-1}(y_{i+1} + y_i) \times (x_i y_{i+1} - y_i x_{i+1})}{3A(P)}$$

O resultado pode ser facilmente implementado em um algoritmo com complexidade $O(n)$, que naturalmente pode fornecer ao mesmo tempo a área do polígono (Programa 1.17).

```
função centróidePolígono(Poligono P): Ponto
início
      real parcela, xc = 0, yc = 0;
      inteiro i, i1;
      Ponto pt;

      para i = 0 até i < P.numVértices faça
      início
            i1 = ((i + 1) mod P.numVértices);

            parcela = ((P.fronteira[i].x * P.fronteira[i1].y) -
                       (P.fronteira[i].y * P.fronteira[i1].x));

            S = S + parcela;
            xc = xc + (P.fronteira[i].x +P.fronteira[i1].x) * parcela;
            yc = yc + (P.fronteira[i].y +P.fronteira[i1].y) * parcela;
      fim;

      se (S != 0) então início
            xc = xc / (3 * S);
            yc = yc / (3 * S);
      fim senão início
            xc = 0;
            yc = 0;
            sinalizar erro e retornar;
      fim;

      S = S / 2;
      pt.x = xc;
      pt.y = yc;
      retorne (pt);
fim.
```

Programa 1.17 - Função `centróidePolígono`

Mas apesar da simplicidade do processo, não existe garantia de que o centróide será um ponto *pertencente* ao polígono. Caso seja necessário encontrar um ponto interno a um polígono simples dado, pode-se utilizar o seguinte processo, que busca precisamente identificar rapidamente uma diagonal do polígono [ORou94]:

- identificar um vértice convexo v_i (por exemplo, o vértice inferior mais à direita, conforme demonstrado na seção 1.1.5.1)
- para cada outro vértice v_j do polígono verificar:

    - se v_j estiver dentro do triângulo $v_{i-1}v_iv_{i+1}$, então calcular a distância v_iv_j
    - armazenar v_j em q se esta distância for um novo mínimo
- ao final do processo, se algum ponto interior a $v_{i-1}v_iv_{i+1}$ for encontrado, então o ponto médio do segmento qv_i é interior ao polígono; se nenhum ponto interior for encontrado, então o ponto médio do segmento $v_{i-1}v_{i+1}$ (ou mesmo o centróide do triângulo $v_{i-1}v_iv_{i+1}$) é interior ao polígono.

Curiosamente, parte da literatura de SIG e mesmo a nomenclatura adotada por alguns sistemas considera uma definição alternativa de centróide, em que mesmo se situa "aproximadamente no centro do polígono" [LaTh92]. Assim, o centróide pode ser determinado por diversos processos, como o centro do retângulo envolvente mínimo, o centro de um círculo inscrito ou circunscrito ao polígono, ou mesmo definido intuitivamente pelo usuário. Uma forma freqüentemente usada para determinar um centróide consiste em simplesmente obter a média aritmética das coordenadas *x* e *y* dos vértices. Embora menos computacionalmente intensivo do que o método apresentado nesta seção, o processo da média tem seus resultados afetados por características dos objetos, ou mesmo pelo processo de digitalização dos polígonos. Como se pode perceber na Figura 1.39, a existência, por alguma razão, de uma concentração de vértices em uma região do polígono causa um deslocamento indesejável do centróide. O deslocamento ocorre justamente em direção à região com maior densidade de vértices, o que pode prejudicar aplicações simples, como o posicionamento de textos gráficos.

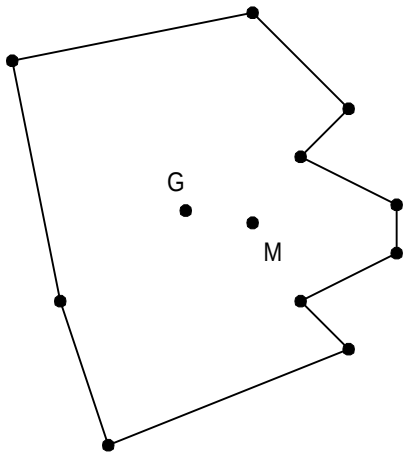


Figura 1.39 - Centróides calculados pela média (M) e como centro de gravidade (G)

1.1.5.4 Ponto em polígono

Um dos problemas mais importantes em SIG consiste em determinar se um dado ponto está dentro ou fora de um polígono. É resolvido, por exemplo, várias vezes cada vez que se usa o botão do *mouse* para selecionar um objeto na tela. Embora a solução seja relativamente simples, é necessário tomar muito cuidado com casos especiais e problemas numéricos, fazendo com que uma implementação realmente robusta seja relativamente difícil de alcançar.

O princípio fundamental para determinar se um ponto *Q* está ou não dentro de um polígono *P* consiste em traçar, a partir de *Q*, uma semi-reta em uma direção qualquer. Quando esta semi-reta intercepta o polígono um número ímpar de vezes, então o ponto

está dentro do polígono; caso contrário, ou seja, caso exista um número par de interseções, o ponto está fora (Figura 1.40).

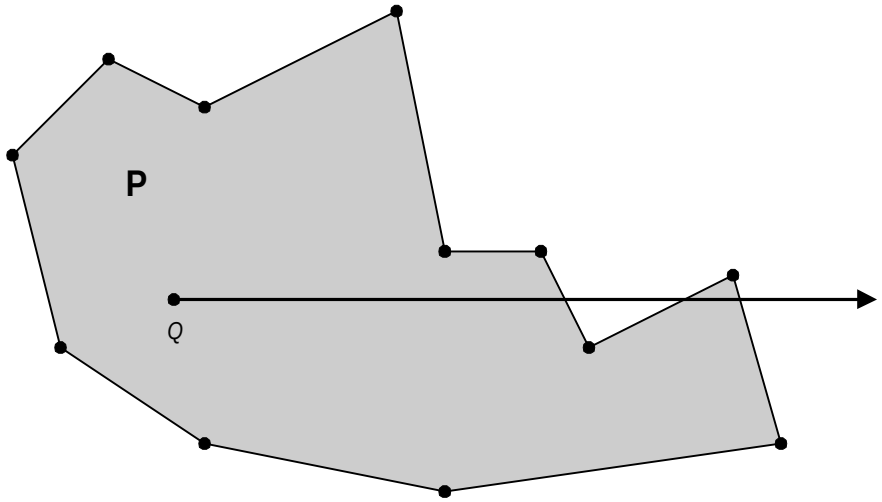


Figura 1.40 - Ponto em polígono

Por conveniência, adota-se em geral uma semi-reta paralela ao eixo dos *x* passando por *Q* e se estendendo para a direita. A contagem do número de interseções pode ser feita com facilidade, verificando quantos segmentos têm um ponto extremo acima e outro abaixo de y_Q. Destes, é preciso verificar em quantos a interseção com a semi-reta ocorre em um ponto de abscissa maior que x_Q. A maior dificuldade está no tratamento de casos degenerados, como:

- a semi-reta passa por uma aresta do polígono (Figura 1.41a);
- a semi-reta passa por um vértice do polígono (Figura 1.41b);
- o ponto *Q* está sobre a fronteira do polígono (Figura 1.41c);
- o ponto *Q* coincide com um vértice do polígono (Figura 1.41d).

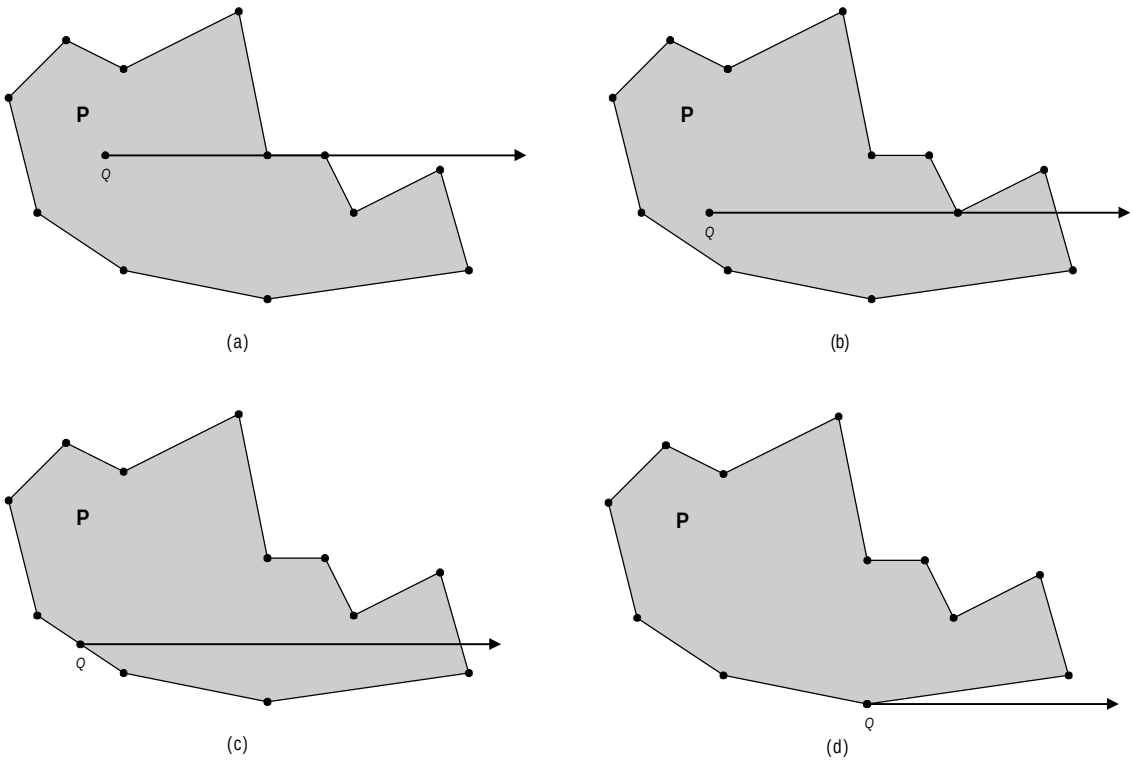


Figura 1.41 - Ponto em polígono - casos degenerados

Para estes casos, a solução está em adotar um critério para a contagem de interseções de modo que:

- se a reta passa por um vértice, a interseção deve ser considerada apenas se for o vértice com maior ordenada do segmento, e ignorada caso contrário;
- se a reta passa por um segmento do contorno do polígono, nenhuma interseção deve ser considerada;
- se o ponto *Q* pertence a um segmento do contorno (exceto pontos extremos), considerar como uma interseção.

O caso em que *Q* coincide com um vértice pode ser tratado pelo primeiro critério. O terceiro critério faz com que todos os pontos da fronteira sejam considerados como pertencentes ao polígono.

```
função pontoEmRetângulo(Ponto Q, Ponto A, Ponto B): booleano
/* testa se Q está contido no retângulo definido por A e B */
início
      Ponto C, D;

      C.x = min(A.x, B.x);
      C.y = min(A.y, B.y);
      D.x = max(A.x, B.x);
      D.y = max(A.y, B.y);

      retorne ((Q.x >= C.x) e (Q.x <= D.x) e
               (Q.y >= C.y) e (Q.y <= D.y));
fim.
```

Programa 1.18 - Função `pontoEmRetângulo`

Assim, pode-se construir um algoritmo (Programa 1.19), que claramente tem complexidade $O(n)$, sendo n o número de vértices do polígono. Observe-se que foi incluído um teste de rejeição rápida (Programa 1.18), comparando o ponto com o retângulo envolvente mínimo do polígono, considerando que este tenha sido previamente determinado e armazenado. Este teste, que tem complexidade $O(1)$, pode representar um ganho de desempenho importante para SIG, onde `pontoEmPolígono` é freqüentemente usada.

```
função pontoEmPolígono(Ponto Q, Poligono P): booleano
início
      inteiro i, numInterseções = 0;
      Ponto C, D;

      se (não (pontoEmRetângulo(Q, P.REMp1, P.REMp2))
      então retorne falso;

      para i = 0 até i < P.numVértices faça
      início
            C = P.fronteira[i];
            D = P.fronteira[i + 1];

            se (C.y != D.y)
            então início
                  calcular interseção;
                  se (interseção for à direita de Q)
                  então numInterseções = numInterseções + 1;
            fim então;
      fim;
      retorne ((numInterseções mod 2) != 0);
fim.
```

Programa 1.19 - Função `pontoEmPolígono`

O'Rourke [ORou94] propõe uma variação em que a interseção é considerada apenas se um dos pontos extremos do segmento da fronteira estiver estritamente acima ou abaixo da semi-reta, enquanto o outro pode estar do lado oposto ou sobre a semi-reta (Programa 1.20[11]). Desta forma, é intencionalmente forçada uma situação em que pontos

[11] O Programa 1.20, embora use os mesmos critérios de solução de casos degenerados, difere do apresentado em [ORou94], pois não realiza a mudança de eixos proposta (que pode ser inconveniente), e procura tirar partido das funções desenvolvidas na seção anterior.

pertencentes a determinados segmentos da fronteira são considerados como pertencentes ao polígono, enquanto outros são considerados como não pertencentes. Como exemplo, considere-se um retângulo com lados paralelos aos eixos. Pelo algoritmo do Programa 1.20, pontos nas laterais esquerda e inferior são considerados dentro, e pontos nas laterais direita e superior são considerados fora do polígono. Este critério é o mais recomendável em situações de particionamento do plano, em que se deseja que qualquer ponto pertença a exatamente um polígono. Pelo critério anterior, pontos sobre a fronteira comum entre dois polígonos pertenceriam a ambos.

```
função pontoEmPolígono1(Ponto Q, Poligono P): booleano
início
      inteiro i, i1, numInterseções = 0;

      se (não (pontoEmRetângulo(Q, P.REMp1, P.REMp2))
      então retorne falso;

      para i = 0 até i < P.numVértices faça
      início
            i1 = i + n - 1;

            se (((P.fronteira[i].y > Q.y) e
                 (P.fronteira[i1].y <= Q.y)) ou
                ((P.fronteira[i1].y > Q.y) e
                 (P.fronteira[i].y <= Q.y)))
            então início
                  calcular interseção;
                  se (interseção for à direita de Q)
                  então numInterseções = numInterseções + 1;
            fim então;
      fim;
      retorne ((numInterseções mod 2) != 0);
fim.
```

Programa 1.20 - Função `pontoEmPolígono1` (variação de `pontoEmPolígono`)

A aplicação do teste de ponto em polígono em SIG depende também do modelo adotado para representar entidades de área. Caso apenas sejam permitidos polígonos simples, o teste conforme apresentado poderá ser usado. No entanto, caso o modelo do SIG permita a existência de regiões formadas por vários polígonos simples isolados, e formando ilhas e buracos, o teste precisará ser aperfeiçoado. Uma possibilidade é testar a inclusão do ponto em todos os polígonos simples que compõem a região, verificando os seguintes casos:

- Ponto contido em apenas um polígono: neste caso, o polígono só poderá ser uma ilha, e portanto o ponto está dentro da região. Caso o polígono seja um buraco, existe erro topológico.

- Ponto contido em mais de um polígono: se o número de ilhas for igual ao número de buracos, o ponto está fora da região (Figura 1.42b); se o número de ilhas for maior que o número de buracos, o ponto está dentro da região (Figura 1.42a). O caso de se ter o número de buracos superior ao número de ilhas indica um erro topológico.

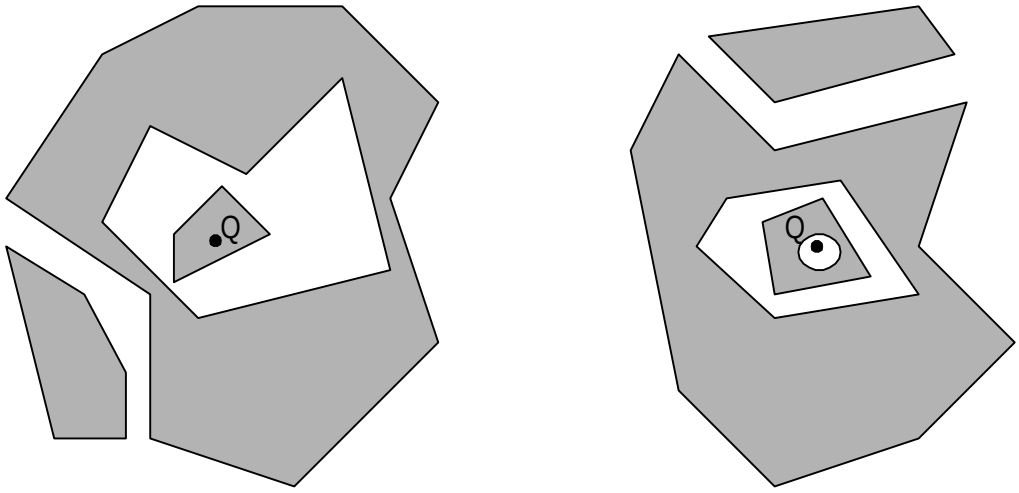


Figura 1.42 - Ponto em região

Para o teste em regiões, conforme descrito acima, recomenda-se usar o Programa 1.20, para evitar conflitos em situações em que o ponto fique na fronteira entre dois ou mais polígonos, o que pode mascarar o resultado. Em qualquer caso, considera-se que não existe superposição entre os polígonos que compõem a região.

1.1.5.5 Interseção, união e diferença de polígonos

Operações sobre polígonos são de fundamental importância em SIG. Através da detecção e processamento da união, interseção e diferença de polígonos, diversos tipos de operações, conhecidas como em conjunto como *polygon overlay*, são viabilizadas. São operações fundamentais para análise espacial, usadas em situações em que é necessário combinar ou comparar dados colocados em camadas distintas. Por exemplo, considere-se uma consulta como "identificar fazendas em que mais de 30% da área é de latossolo roxo". Para executar esta análise, é necessário combinar uma camada de objetos poligonais (os limites de propriedades rurais) com outra (o mapa de tipos de solo), para obter uma nova camada, de cujo conteúdo podem ser selecionados diretamente os objetos que atendem ao critério de análise colocado.

Algumas vezes, o *polygon overlay* é definido como uma operação topológica, ou seja, que é executada sobre dados organizados em uma estrutura de dados topológica. Estas estruturas e operações sobre elas serão apresentadas na seção 1.2. As funções de processamento de polígonos que serão descritas a seguir são utilizadas em sistemas não topológicos, ou em situações em que o processamento é feito de maneira isolada, como na criação e uso de *buffers* (vide seção 1.1.5.6).

Este problema é também estudado com bastante intensidade em computação gráfica, onde é denominado *recorte de polígonos*. O caso mais simples é o recorte de um polígono por um retângulo, correspondendo à situação corriqueira de apresentação em tela. O caso geral é o recorte de um polígono por outro, ambos genéricos (não convexos), que ocorre no processamento de remoção de superfícies escondidas, em síntese de imagens 3D. Outro uso para o recorte de polígonos é a quebra de objetos em

uma cena para processamento de *ray tracing* em computadores paralelos. Em computação gráfica, o problema pode ainda incluir os casos de polígonos não simples e polígonos com buracos [Vatt92].

Para realizar operações sobre polígonos, é interessante aplicar um passo preliminar de detecção rápida da possibilidade interseção entre os polígonos. Assim, se não for possível que dois polígonos P e Q tenham interseção, então podemos concluir diretamente que $P \cup Q = \{P, Q\}$, $P \cap Q = \varnothing$, $P - Q = P$ e $Q - P = Q$. Uma maneira simples de testar se dois polígonos têm ou não interseção é usar inicialmente o teste de interseção dos retângulos envolventes mínimos (seção 1.1.2.2).

No caso geral, operações de união, interseção ou diferença entre dois polígonos simples podem gerar diversos polígonos como resultado. Mais ainda, os polígonos resultantes poderão conter buracos. A Figura 1.43 contém exemplos de produção de múltiplos polígonos e de polígonos com buracos em operações de interseção, união e diferença.

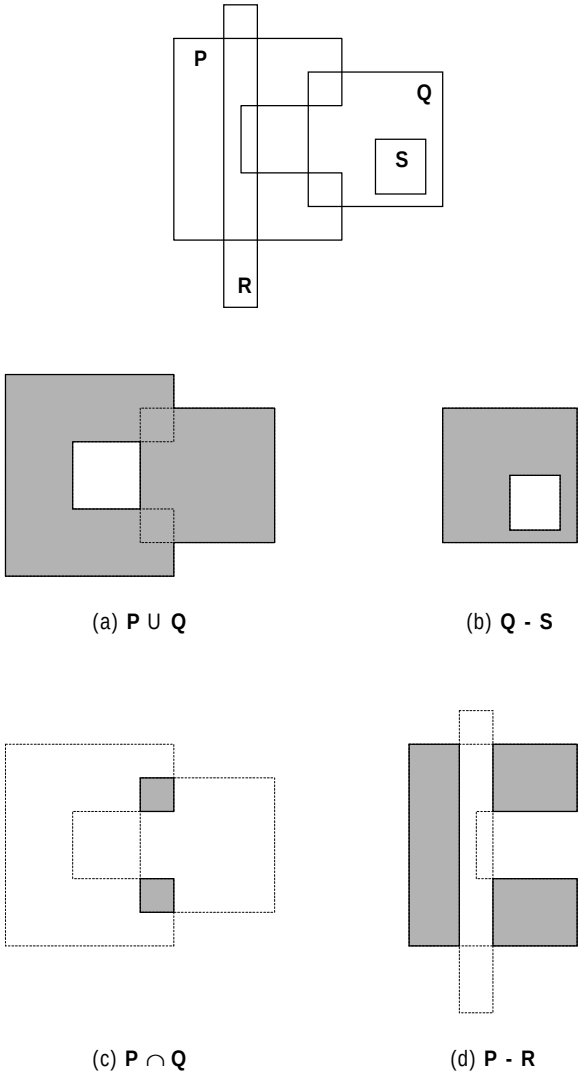


Figura 1.43 - Operações sobre polígonos produzindo buracos e múltiplos polígonos

Existe uma proposta de realizar a análise de interseção e de união entre polígonos usando a técnica de varredura do plano, de maneira semelhante à interseção de *n*

segmentos. Esta técnica consiste em dividir os polígonos em fatias horizontais, usando linhas que passam pelos vértices e pelos pontos de interseção entre arestas, ordenando os pontos segundo o eixo das abscissas e fazendo a varredura do plano [NiPr82]. Cada fatia é composta de um ou mais trapezóides, provenientes de um ou de ambos os polígonos, sendo relativamente simples detectar sua interseção em cada fatia [LaTh92].

No entanto, um método mais interessante foi apresentado por Margalit e Knott [MaKn89], e permite realizar operações de interseção, união e diferença em tempo $O(n \log n)$, e partindo de polígonos que podem ter ilhas ou buracos. O algoritmo é sensível à orientação dos polígonos, e exige que os vértices de ilhas sejam codificados em um sentido (por exemplo, anti-horário) e os vértices de buracos sejam dispostos no sentido inverso (horário). Isto coincide com a convenção usada para calcular a área de polígonos, conforme apresentado na seção 1.1.5.2.

Tabela 1.5 - Orientação dos polígonos de entrada de acordo com a operação

Polígonos		**Operações**			
P	***Q***	$P \cap Q$	$P \cup Q$	$P - Q$	$Q - P$
ilha	**ilha**	*manter*	*manter*	*inverter*	*inverter*
ilha	**buraco**	*inverter*	*inverter*	*manter*	*manter*
buraco	**ilha**	*inverter*	*inverter*	*manter*	*manter*
buraco	**buraco**	*manter*	*manter*	*inverter*	*inverter*

O algoritmo tem seis passos, que serão descritos a seguir.

1. **Normalizar a orientação** dos polígonos de entrada *P* e *Q*, e inverter a orientação de *Q* dependendo do tipo de operação e da natureza (ilha ou buraco) dos dois polígonos de entrada, de acordo com a Tabela 1.5.

2. **Classificar os vértices**, verificando se cada um está *dentro*, *fora* ou *na fronteira* do outro polígono, usando o teste de ponto em polígono (seção 1.1.5.4). Inserir os vértices assim classificados em duas listas circulares, *PL* e *QL*, onde aparecerão em seqüência, de modo a definir as arestas por adjacência.

3. **Encontrar as interseções** entre arestas dos dois polígonos, usando o teste de interseção de *n* segmentos (seção 1.1.3). Inserir os pontos de interseção na posição apropriada em *PL* e *QL*, classificando-os como *na fronteira*. A partir deste ponto, teremos um conjunto de *fragmentos de arestas* em lugar das arestas originais. É necessário cuidar do caso especial de interseção ao longo de uma aresta comum, ou parte dela. Neste caso, ambos os pontos extremos da aresta devem ser classificados como *na fronteira* e inseridos nas listas.

4. **Classificar os fragmentos de arestas** (definidos pelos pares de vértices) formados em *PL* e *QL* com relação ao outro polígono, entre *interior*, *exterior* ou na *fronteira*.

Não é necessário realizar novamente o teste de ponto em polígono. Uma aresta pode ser considerada *interior* ao outro polígono caso pelo menos um de seus vértices esteja classificado como *dentro*. Da mesma forma, uma aresta pode ser classificada como *exterior* ao outro polígono caso pelo menos um de seus vértices esteja classificado como *fora*. Se ambos os vértices estiverem classificados como *na fronteira*, então é necessário verificar a situação de um ponto interno ao segmento (por exemplo, seu ponto médio). Se este ponto estiver fora do outro polígono, então a aresta é classificada como *exterior*. Se o ponto estiver dentro do outro polígono, então a aresta é classificada como *interior*. Se o ponto estiver na fronteira, a aresta é classificada como *fronteira*.

Arestas na fronteira constituem um caso degenerado, que requer tratamento especial. Se existe um fragmento de aresta na fronteira de *P*, então necessariamente existe também um na fronteira de *Q*. Estes fragmentos podem estar orientados na mesma direção ou em direções opostas. A implementação pode decidir o que fazer nestes casos, ou seja, se interseções com dimensão de segmento ou de ponto serão ou não retornadas. Se as interseções como segmento forem retornadas, serão formadas por um ciclo de duas arestas sobrepostas, cada uma em uma direção. Interseção em um ponto será retornada como um ciclo de duas arestas, cada uma em uma direção, ligando dois vértices sobrepostos. Desta forma preserva-se a topologia do resultado (sempre cadeia fechada de segmentos), mas em SIG é mais interessante detectar estes casos e retornar objetos da dimensão adequada (no caso, ponto)[12].

5. **Selecionar e organizar as arestas** para formar os polígonos de resultado. Este processo de seleção é baseado na combinação das duas listas em uma, denominada *RL*, usando apenas as arestas que interessam para a operação, conforme definido na Tabela 1.6.

6. **Construir os polígonos de resultado**, selecionando uma aresta e, com base em seu ponto final, procurar em *RL* sua continuação, até fechar o polígono. Repetir o processo, eliminando de *RL* a cada passo as arestas utilizadas, até que *RL* fique vazia.

[12] Para uma análise mais completa, inclusive com as combinações de hipóteses nos casos de ilhas e buracos, vide [MaKn89].

Tabela 1.6 - Tipos de arestas para seleção de acordo com o tipo de operação e os tipos de polígonos de entrada

Polígonos		**Operações**							
		$P \cap Q$		$P \cup Q$		$P - Q$		$Q - P$	
P	***Q***	*P*	*Q*	*P*	*Q*	*P*	*Q*	*P*	*Q*
ilha	**buraco**	interior	interior	exterior	exterior	exterior	interior	interior	exterior
ilha	**buraco**	exterior	interior	interior	exterior	interior	interior	exterior	exterior
buraco	**ilha**	interior	exterior	exterior	interior	exterior	exterior	interior	interior
buraco	**buraco**	exterior	exterior	interior	interior	interior	exterior	exterior	interior

Os polígonos resultantes manterão a orientação adotada para ilhas e buracos, o que é bastante conveniente.

O exemplo a seguir mostra o funcionamento do algoritmo para realizar operações sobre os polígonos *P* e *Q*, apresentados na Figura 1.44. Ambos os polígonos são ilhas, e portanto os vértices estão dispostos em sentido anti-horário. Como se pode perceber visualmente, a interseção destes polígonos é formada por dois polígonos isolados, e a união é um polígono com um buraco. Os vértices de cada polígono estão numerados da maneira usual, e os pontos de interseção são identificados como i_1, i_2, i_3 e i_4.

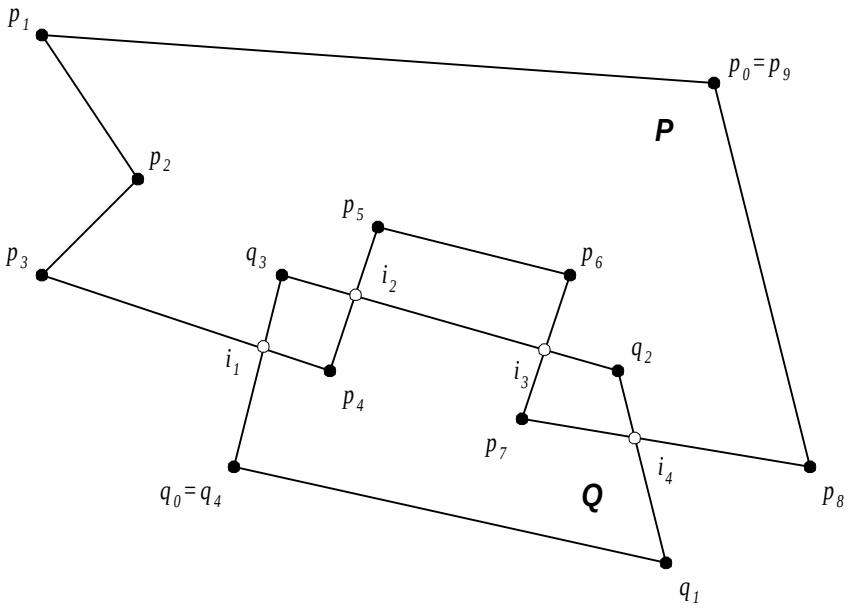


Figura 1.44 - Interseção de polígonos

De acordo com a Tabela 1.5, para realizar operações de união e interseção sobre dois polígonos ilha, não é necessário inverter a orientação dos polígonos *P* e *Q* dados. O passo seguinte consiste em classificar os vértices e formar as listas *PL* e *QL*, que ficam:

	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
PL	**f**	**f**	**f**	**f**	**d**	**f**	**f**	**d**	**f**	**f**

	q_0	q_1	q_2	q_3	q_4
QL	**f**	**f**	**d**	**d**	**f**

Em seguida, os pontos de interseção i_1, i_2, i_3 e i_4 são determinados e inseridos em *PL* e *QL*, que ficam agora:

	p_0	p_1	p_2	p_3	i_1	p_4	i_2	p_5	p_6	i_3	p_7	i_4	p_8	p_9
PL	f	f	f	f	**nf**	d	**nf**	f	f	**nf**	d	**nf**	f	f

	q_0	q_1	i_4	q_2	i_3	i_2	q_3	i_1	q_4
QL	f	f	**nf**	d	**nf**	**nf**	d	**nf**	f

O passo de classificação das arestas vem a seguir, analisando os pares de vértices na seqüência encontrada nas listas *PL* e *QL*, definindo arestas interiores, exteriores ou de fronteira, a partir da classificação dos vértices. O resultado é organizado na lista *RL*, que fica:

	p_0	p_1	p_2	p_3	i_1	p_4	i_2	p_5	p_6	i_3	p_7	i_4	p_8	q_0	q_1	i_4	q_2	i_3	i_2	q_3	i_1
	p_1	p_2	p_3	i_1	p_4	i_2	p_5	p_6	i_3	p_7	i_4	p_8	p_9	q_1	i_4	q_2	i_3	i_2	q_3	i_1	q_4
RL	**e**	**e**	**e**	**e**	**i**	**i**	**e**	**e**	**e**	**i**	**i**	**e**	**e**	**e**	**e**	**i**	**i**	**e**	**i**	**i**	**e**

Observe-se a classificação da aresta i_3i_2, que somente pôde ser feita analisando a posição do seu ponto médio em relação ao polígono *P*.

A partir de *RL* é possível construir os polígonos resultantes da interseção de *P* e *Q*. De acordo com a Tabela 1.6, para a união selecionamos as arestas exteriores:

	p_0	p_1	p_2	p_3	i_2	p_5	p_6	i_4	p_8	q_0	q_1	i_3	i_1
	p_1	p_2	p_3	i_1	p_5	p_6	i_3	p_8	p_9	q_1	i_4	i_2	q_4
$P \cup Q$	**e**	**e**	**e**	**e**	**e**	**e**	**e**	**e**	**e**	**e**	**e**	**e**	**e**

Para a interseção, ainda de acordo com a Tabela 1.6, selecionamos as arestas interiores:

	i_1	p_4	i_3	p_7	i_4	q_2	i_2	q_3
	p_4	i_2	p_7	i_4	q_2	i_3	q_3	i_1
$P \cap Q$	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**

Para compor o resultado da união selecionamos por exemplo a primeira aresta, p_0p_1, e a partir daí buscamos continuações até voltar a p_0. O resultado inicial é, portanto, o polígono cujos vértices são p_0, p_1, p_2, p_3, i_1, q_0 (q_4), q_1, i_4, p_8, p_0 (p_9). No entanto, resta ainda na lista outro polígono, com vértices i_2, p_5, p_6, i_3, i_2. Os vértices do primeiro polígono estão em sentido anti-horário, formando portanto uma ilha. O segundo polígono está em sentido horário, e é portanto um buraco. Analogamente, na interseção obtemos os polígonos i_1, p_4, i_2, q_3, i_1 e i_3, p_7, i_4, q_2, i_3. Neste caso, ambos os polígonos obtidos estão em sentido anti-horário, e portanto são ambos ilhas.

Analisando a complexidade do algoritmo, sendo n_P e n_Q o número de vértices ds polígonos originais, e n_i o número máximo de interseções, verifica-se que:

1. Encontrar e, se necessário, inverter a orientação de cada polígono custa $O(n_P + n_Q)$.

2. Classificar os vértices custa $O(n_Q)$ em P e $O(n_P)$ em Q. Inserir nas listas PL e QL custa $O(1)$, e portanto o custo total do passo é $O(n_P + n_Q)$.

3. Encontrar todas as interseções entre as arestas de P e Q custa, conforme verificado na seção 1.1.3, $O((n_P + n_Q + n_i) \log(n_P + n_Q))$[13]. Inserir cada uma destas interseções em ordem nas listas PL e QL custa $O(n_i)$, gerando um custo total de $O(n_i^2)$. Portanto, a complexidade total deste passo é $O((n_P + n_Q + n_i) \log(n_P + n_Q) + n_i^2)$.

4. Classificar cada aresta pode ser feito em tempo constante, se um dos pontos extremos for um vértice original de P ou de Q. Se ambos os pontos extremos estão na fronteira, é necessário executar o procedimento de ponto em polígono, ao custo de $O(n_P)$ para fragmentos de aresta provenientes de P, ou $O(n_Q)$ no caso inverso. O custo total é, portanto, de $O(n_P + n_Q + n_i \,.\, \max(n_P, n_Q))$

5. Selecionar as arestas que interessam tem custo proporcional ao número de fragmentos de aresta produzidos, ou seja, $O(n_P + n_Q + n_i)$. Estas arestas podem ser organizadas de forma conveniente, de modo a facilitar a execução do último passo, segundo uma lista ordenada pelo vértice inicial. A ordenação custa $O((n_P + n_Q + n_i) \log(n_P + n_Q + n_i))$, que é o custo total deste passo.

6. Construir os polígonos resultantes depende da eficiência das pesquisas na estrutura de dados gerada no passo anterior. No caso de lista ordenada, o custo de localização do sucessor é $O(\log(n_P + n_Q + n_i))$, utilizando pesquisa binária. O custo total é, portanto, $O((n_P + n_Q + n_i) \log(n_P + n_Q + n_i))$.

---

[13] Neste ponto da análise, [MaKn89] considera um algoritmo força-bruta para encontrar as interseções, consumindo tempo $O(n_P.n_Q)$. Naturalmente, o algoritmo apresentado na seção 1.1.3 é mais eficiente, baixando a complexidade total deste passo.

Compondo os custos de cada passo, chega-se à complexidade deste algoritmo no pior caso, que é dominada pelo passo 3, de inserção dos pontos de interseção em uma lista. Como, no pior caso, tem-se $n_i = n_P \,.\, n_Q$, chega-se à complexidade $O((n_P \,.\, n_Q)^2)$. Nos casos mais usuais em SIG, no entanto, o número de interseções em geral fica muito abaixo de $n_P \,.\, n_Q$, e portanto espera-se um desempenho prático bastante superior ao de pior caso. Para uma análise mais detalhada desta complexidade, bem como considerações a respeito do consumo de espaço, vide [MaKn89].

1.1.5.6 Criação de *buffers*

Um dos recursos mais úteis em SIG é a capacidade de gerar polígonos que contornam um objeto a uma determinada distância. Sua principal função é materializar os conceitos de "perto" e "longe", embora sem o componente *fuzzy* que caracteriza o raciocínio humano nestes termos. Um exemplo de consulta que demanda a utilização de *buffers*: "localizar todos os postos de gasolina ao longo de uma rodovia". Considerando que os postos estejam representados por um ponto, e a rodovia esteja representada pela linha de seu eixo, é preciso encontrar uma distância que materialize o conceito de "ao longo", por exemplo 100 metros. O processamento é feito construindo um polígono que contenha todos os pontos do plano localizados a 100 metros do eixo da rodovia ou menos. Em seguida, determina-se quais postos de gasolina estão contidos neste polígono.

Buffers podem ser construídos ao redor de qualquer tipo de objeto geográfico vetorial: pontos, linhas ou polígonos. No caso de pontos, o *buffer* é simplesmente um círculo cujo raio é a distância desejada. Em linhas e polígonos, o *buffer* é construído a partir da união de *buffers* elementares, que são construídos ao redor de cada segmento e cada vértice. Estes *buffers* elementares são simplesmente círculos, ao redor dos vértices, e retângulos ao redor dos segmentos, com lados paralelos a estes (Figura 1.45).

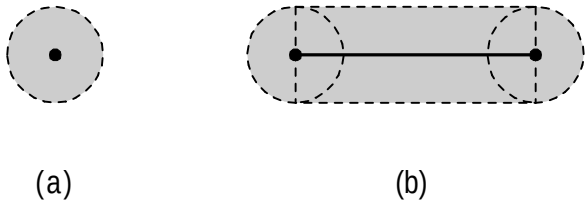


Figura 1.45 - *Buffers* elementares ao redor de ponto (a) e segmento (b)

O *buffer* final é exatamente a união dos *buffers* elementares que foram construídos (Figura 1.46). No caso de polígonos, a união deve ser feita também com o próprio polígono, para que todos os pontos interiores a ele sejam considerados (Figura 1.47).

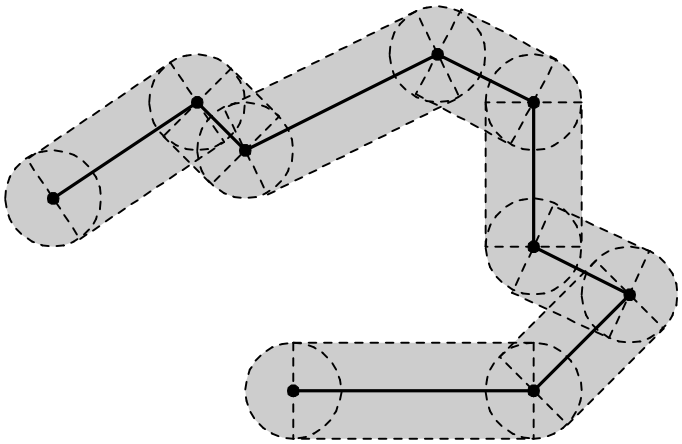


Figura 1.46 - *Buffer* ao redor de linha

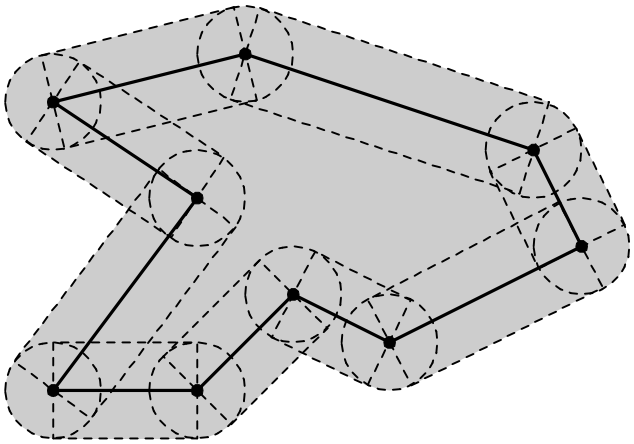


Figura 1.47 - *Buffer* ao redor de polígono

Ocorre também com freqüência a necessidade de se ter *buffers* com distância *negativa*, ou seja, áreas que são construídas no interior de polígonos a uma dada distância da fronteira. Este tipo de *buffer* é conhecido como *skeleton*, ou como *setback*. O processamento é diferente apenas em um ponto: constrói-se a união dos *buffers* elementares ao longo da fronteira, considerando o valor absoluto da distância fornecida, e em seguida determina-se a diferença entre o polígono inicial e a união dos *buffers* (Figura 1.48).

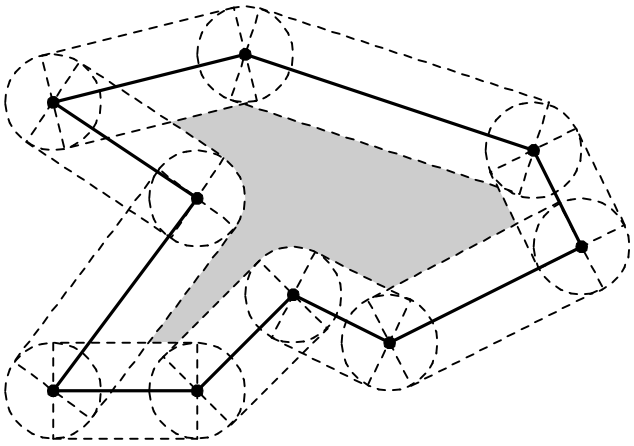


Figura 1.48 - *Buffer* "negativo" de polígono

Observe-se que, conforme foi alertado na seção 1.1.5.5, o resultado de uma operação de criação de *buffers* pode ser formado por vários polígonos separados, ou por polígonos com buracos (Figura 1.49).

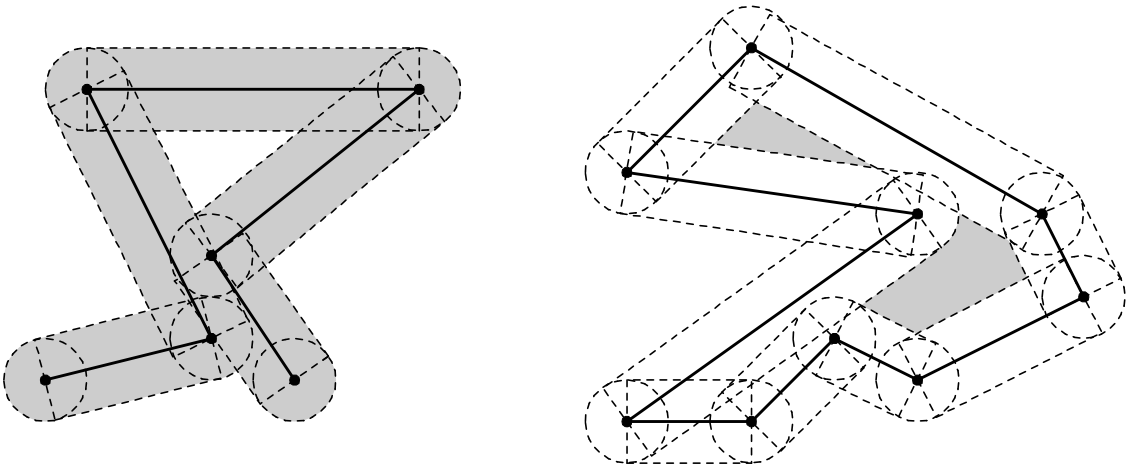


Figura 1.49 - Buffers com buracos (a) e formados por polígonos isolados (b)

1.1.6 Fecho convexo

O fecho convexo é uma das estruturas mais usadas em geometria computacional. Tem utilidade em diversas situações, na geração de outras estruturas e como apoio a algoritmos mais complexos. Exemplos do uso de fechos convexos podem ser encontrados em robótica, reconhecimento de padrões, processamento de imagens, e até mesmo em otimização.

Em SIG, o fecho convexo pode ser útil para delimitar regiões definidas por um conjunto de pontos, como por exemplo a área afetada por alguma doença, cujos casos são individualmente lançados como pontos no mapa. Também é útil na construção de limites para triangulações, do tipo encontrado em modelagem digital de terrenos (vide seção 1.1.7.4).

O conceito de fecho convexo de um conjunto *S* de pontos é bastante simples. Uma imagem freqüentemente invocada nos livros de geometria computacional é a de um elástico (a fronteira do fecho) colocado ao redor de todos os pregos (pontos) dispostos sobre uma tábua (o plano). Este conceito pode ser naturalmente estendido para três ou mais dimensões, mas estaremos interessados particularmente em fechos convexos no plano, por se tratar da variação mais útil em SIG.

Numa definição mais formal, o fecho convexo de um conjunto *S* de pontos é o menor conjunto convexo que contém *S*. Um conjunto convexo *C* é aquele em que $\forall P_1, P_2 \in C \longrightarrow \overline{P_1 P_2} \subset C.$ Geometricamente, o fecho convexo é um polígono convexo cujos vértices são um subconjunto de *S*, e que contém todos os pontos de *S*.

O'Rourke [ORou94] lista uma série de definições alternativas e equivalentes do fecho convexo, as mais interessantes das quais estão relacionadas a seguir:

- O fecho convexo de um conjunto *S* de pontos do plano é o menor polígono convexo *P* que contém *S*, sendo "menor" entendido no sentido de que não existe outro polígono *P'* tal que $P \supset P' \supseteq S$.

- O fecho convexo de um conjunto *S* de pontos é a interseção de todos os conjuntos convexos que contêm *S*.

- O fecho convexo de um conjunto *S* de pontos do plano é o polígono convexo envolvente *P* que tem a menor área.

- O fecho convexo de um conjunto *S* de pontos do plano é a união de todos os triângulos determinados por pontos pertencentes a *S*.

O problema de encontrar o fecho convexo pode ser colocado de duas maneiras: (1) construir a fronteira do polígono *FC*(*S*) (fecho convexo de S), ou (2) identificar os pontos de *S* que são vértices de *FC*(*S*). Foi demonstrado que ambas as variações têm a mesma complexidade, apesar de parecer, instintivamente, que a segunda alternativa seria mais fácil, uma vez que não exige a construção das ligações entre os vértices (ou seja, não exige a determinação da *seqüência* de vértices) para formar o polígono.

Foi demonstrado também que o problema clássico de ordenação pode ser transformado, em tempo linear, no problema de encontrar o fecho convexo [PrSh88]. A conseqüência disso é que um algoritmo ótimo para determinar o fecho convexo tem complexidade computacional $\Omega(n \log n)$ sobre o número de pontos em *S*.

1.1.6.1 Pontos interiores e arestas extremas

Um algoritmo imediato para determinar quais seriam os vértices do fecho convexo pode ser construído da seguinte maneira [ORou94]:

1. para cada combinação de três pontos (p_i, p_j, p_k) de *S*, construir um triângulo;

2. testar cada um dos outros *n* - 3 pontos contra este triângulo, usando o procedimento `pontoEmTriângulo` (seção 1.1.2.1);

3. marcar como “interior” todo ponto que estiver dentro do triângulo;

4. ao final do processamento, os pontos não marcados serão os vértices do fecho convexo.

Este algoritmo é claramente $O(n^4)$, uma vez que depende da combinação dos pontos três a três, posteriormente testando cada um dos n - 3 pontos contra o triângulo.

Uma alternativa um pouco menos custosa consiste em procurar determinar as arestas do fecho convexo, da seguinte maneira [ORou94]:

1. para cada par ordenado (p_i, p_j) de pontos de S, definir uma reta;

2. verificar o posicionamento de todos os demais pontos de S em relação a esta reta;

3. se algum ponto não estiver à esquerda ou sobre a reta p_ip_j (usar o procedimento `lado`, seção 1.1.2.2), então o segmento p_ip_j não é uma aresta do fecho convexo.

4. se todos os demais pontos estiverem à sua esquerda, o segmento p_ip_j é uma aresta do fecho convexo.

Observe-se que o teste tem que ser feito com cada par *ordenado* (p_i, p_j), para que se possa sempre testar se os demais pontos estão especificamente *à esquerda*. Observe-se também que existe a necessidade de um teste adicional, para eliminar a situação em que três vértices alinhados definem duas arestas superpostas para o fecho convexo. Este algoritmo consome tempo $O(n^3)$, um vez que é necessário formar os pares de pontos ($O(n^2)$) e depois testá-los contra os n - 2 pontos restantes. Este algoritmo foi aperfeiçoado por Jarvis, que observou que, uma vez que se tenha uma aresta p_ip_j do contorno convexo, uma aresta subseqüente necessariamente parte de p_j. Com isso, é possível reduzir a complexidade para $O(n^2)$ [PrSh88]. Este algoritmo é conhecido na literatura como *Jarvis’ march*.

1.1.6.2 Quickhull

Um aperfeiçoamento da estratégia de pontos interiores foi proposto independentemente por diversos pesquisadores, e denominado posteriormente *quickhull* por Preparata e Shamos [PrSh88], devido à sua semelhança com o *quicksort*.

O *quickhull* inicialmente determina os pontos extremos de S à esquerda, à direita, acima e abaixo, ou seja, os pontos com as máximas e mínimas ordenadas e abscissas, definindo também um retângulo envolvente mínimo para o conjunto de pontos. Isto pode ser feito em tempo linear. Em seguida, estes quatro pontos são conectados, formando um quadrilátero. Todos os pontos interiores a este quadrilátero podem ser imediatamente descartados, pois são pontos interiores. O problema fica, então, reduzido a encontrar os pontos extremos nos quatro triângulos definidos entre o quadrilátero inicial e o retângulo envolvente mínimo do conjunto de pontos (Figura 1.50).

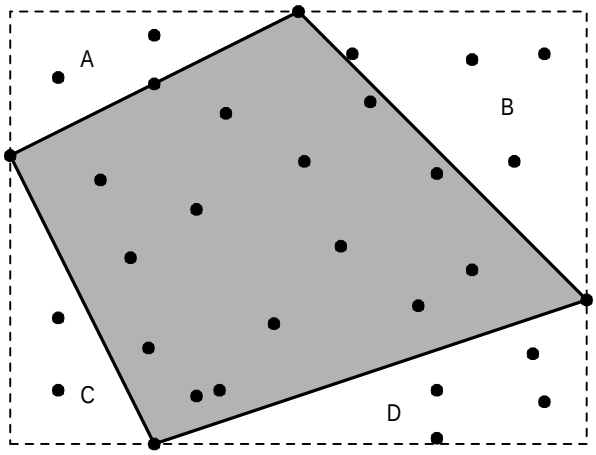


Figura 1.50 - Quickhull - passo inicial

Em cada triângulo, conhece-se dois pontos pertencentes ao fecho convexo. O problema é, então, encontrar qual o ponto contido no triângulo que está mais distante do segmento formado por estes pontos extremos, formando um novo triângulo. Por exemplo, na Figura 1.51, a região triangular *D* definida pelo passo inicial contém quatro pontos. Um triângulo é formado entre os extremos já conhecidos e o mais distante deles, que é naturalmente pertencente ao fecho convexo. Os pontos interiores ao triângulo *D* são descartados, e as novas arestas do triângulo servem de base para a construção de novos triângulos, recursivamente, até que não seja mais encontrado qualquer ponto interior ou exterior.

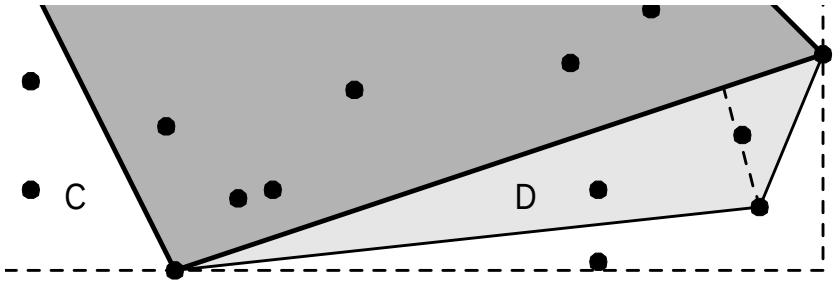


Figura 1.51 - Quickhull - passo recursivo

A análise da complexidade do procedimento recursivo é feita de maneira semelhante à do *quicksort*, admitindo que, a cada passo, o conjunto de pontos é dividido ao meio. A equação de recorrência resultante indica que o algoritmo *quickhull* é, no pior caso, $O(n^2)$, e portanto melhor que os algoritmos anteriores, mas pior que o ótimo esperado, que é $O(n \log n)$ [ORou94].

1.1.6.3 Algoritmo de Graham

Este algoritmo data de 1972, e é considerado um dos pioneiros no campo da geometria computacional. A idéia básica do algoritmo de Graham é bastante simples. Parte-se do

pressuposto que é dado um ponto p_0, interior ao fecho convexo[14], e assume-se que não existam três pontos colineares, incluindo o ponto dado. Em seguida, calcula-se o ângulo com a horizontal, no sentido anti-horário, formado pelo segmento entre o ponto p_0 e todos os demais[15]. Os pontos são ordenados segundo este critério. Os pontos são então examinados um a um segundo essa ordem crescente, em uma *varredura* (daí a denominação usual deste algoritmo, *Graham's scan*, ou varredura de Graham), e o fecho convexo é construído de maneira incremental ao redor de p_0.

A cada passo, o fecho já determinado é correto, considerando apenas o conjunto de pontos examinado até o momento. A avaliação de cada ponto adicional, no entanto, pode fazer com que pontos sejam retirados do fecho, na ordem inversa à em que foram inseridos. Por isso, costuma-se armazenar os pontos do fecho convexo em uma pilha.

A Figura 1.52 mostra um exemplo. Inicialmente, o fecho é constituído dos pontos p_0, p_1 e p_2, sendo p_1 e p_2 colocados na pilha nesta ordem. Em seguida, o ponto p_3 é inserido no fecho, pois p_2p_3 constitui uma curva à esquerda em relação a p_1p_2. Analisando o ponto p_4, o próximo na seqüência ao redor de p_0, verifica-se que p_3p_4 forma uma curva à direita. Assim, elimina-se p_3 do alto da pilha, e testa-se p_2p_4, como agora tem-se uma curva à esquerda, empilha-se p_4 e o algoritmo prossegue de forma incremental. O fechamento se dá quando é analisado o ponto p_{10}, que é ligado ao ponto inicial p_1, fechando a cadeia. O exemplo contém o caso mais simples, em que o primeiro ponto analisado pertence ao fecho convexo. Se ocorresse o contrário, código adicional teria que ser adicionado para retirar o ponto p_1, e possivelmente outros, que estão no fundo da pilha ao final do processamento. Outro problema ocorre quando o p_2 não pertence ao fecho convexo. O teste em p_3 detectaria uma curva à direita a partir de p_1p_2, e portanto, pela lógica apresentada, p_2 teria que ser eliminado da pilha. Ocorre que, nesta situação, a pilha ficaria com apenas um ponto, tornando impossível a análise das curvas à direita e à esquerda.

---

[14] Não é necessário que o ponto p_0 pertença a *S*. Pode-se tomar um ponto interior qualquer, por exemplo o baricentro do conjunto de pontos, ou seja $\frac{1}{n}\sum p_i$ [FiCa91]. Em seu artigo inicial, Graham inclui um método detalhado, de complexidade linear, para escolher tal ponto, analisando trios de pontos até encontrar um não-colinear, e então adotando o baricentro do triângulo como origem [ORou94].

[15] Vide função `lado`, seção 1.1.2.2.

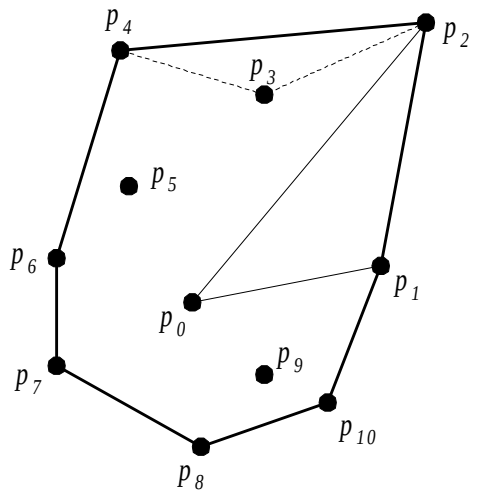


Figura 1.52 - Algoritmo de Graham, origem em ponto interior

Estes problemas podem ser resolvidos com uma escolha mais conveniente da origem, em particular se a origem for um ponto do fecho convexo. O ponto ideal para servir de origem é o ponto inferior do conjunto, necessariamente pertencente ao fecho. Se existirem diversos pontos com a mesma ordenada mínima, toma-se o mais à direita deles. Com isso, o ângulo formado entre p_{n-1}, p_0 e p_1 é necessariamente convexo, e demonstra-se que nesta situação pelo menos os dois primeiros pontos pertencem ao fecho. A pilha é inicializada com p_{n-1} e p_0, e estes pontos nunca serão desempilhados, resolvendo o problema de inicialização descrito anteriormente. Além disso, como p_0 pertence ao fecho, não ocorrerá também o problema de término ao final da varredura dos pontos.

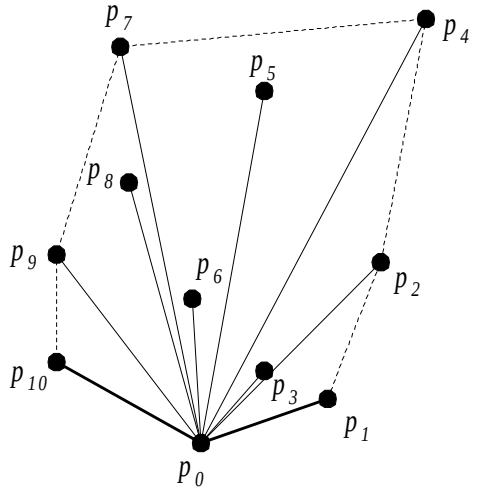


Figura 1.53 - Algoritmo de Graham, origem no ponto inferior mais à direita

Na Figura 1.53, os pontos estão numerados segundo a ordem crescente do ângulo ao redor de p_0. É importante lembrar que a concepção apresentada até o momento se baseia na hipótese de que não se tem três pontos colineares no conjunto, o que não se pode garantir em situações práticas. Portanto, a implementação do algoritmo deve prever recursos para resolver estes casos. Especificamente, é necessário acrescentar um critério

ao algoritmo de ordenação dos pontos ao redor de p_0, dando prioridade em caso de colinearidade para os pontos mais próximos de p_0. Ou seja, considera-se $p_a < p_b$ se os ângulos forem iguais mas $|p_a - p_0| < |p_b - p_0|$. Com esta regra, o ponto mais próximo será eliminado quando o mais distante for analisado. Esta regra também resolve o caso em que existe uma colinearidade nos últimos pontos, fazendo com que o mais próximo seja o penúltimo e o mais distante o último, e portanto pertencente ao fecho convexo conforme desejado.

Um exemplo destas situações degeneradas está apresentado na Figura 1.54. No caso, o ponto p_1 tem que ficar antes de p_2 na ordenação para que seja possível eliminá-lo sem problemas, o que não ocorreria se a situação fosse invertida. Também o ponto p_8 deve vir antes do ponto p_9, pois é p_9 que pertence ao fecho convexo, e que será colocado definitivamente no fundo da pilha dos vértices do fecho.

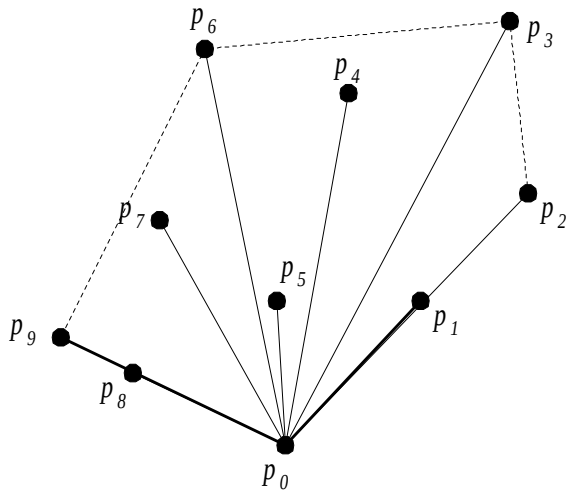


Figura 1.54 - Algoritmo de Graham, situações degeneradas

O Programa 1.21 apresenta o pseudocódigo para a implementação do algoritmo de Graham, utilizando uma pilha conforme descrito, e também usando a função `lado` para fazer o teste da curva à esquerda. Uma característica da implementação usando este recurso é que apenas pontos que são cantos do contorno convexo são retornados. Quaisquer pontos pertencentes ao fecho, mas contidos em alguma aresta, são abandonados. Caso seja interessante ou importante preservar estes pontos, basta alterar a comparação do resultado da função `lado` para permitir a igualdade com zero.

```
função graham(inteiro n, Ponto Q[n]): Pilha;
início
      inteiro i;
      Ponto P[n];
      Pilha S;

      P[0] = ponto de Q inferior e mais à direita;

      ordenar todos os pontos de Q ao redor de P[0] pelo ângulo;
      em caso de empate no ângulo, favorecer o mais próximo de P[0];
      acrescentar os demais pontos de Q a P, como P[1]...P[n-1];

      push(S, (n - 1));
      push(S, 0);
      i = 1;

      enquanto (i < n) faça
      início
            se lado(P[topo(S)], P[topo(topo(S))], P[i]) > 0 então
            início
                  push(S, i);
                  i = i + 1;
            fim se
            senão
                  pop(S);
      fim;

      retorne(S);
fim.
```

Programa 1.21 - Algoritmo de Graham para fecho convexo

A complexidade do algoritmo de Graham pode ser facilmente determinada, observando que o processo de varredura é claramente $O(n)$. O tempo total é portanto dominado pelo passo de ordenação dos pontos ao redor de p_0, que é $O(n \log n)$, correspondendo ao ótimo teórico.

Diversos algoritmos surgiram após o de Graham, utilizando processos diversos para atingir resultados também $O(n \log n)$, mas apresentando características que facilitam a extensão para dimensões mais altas. Existe também um algoritmo que usa a técnica de dividir para conquistar [PrSh88], usando um enfoque parecido com o do algoritmo *merge sort* (*Merge Hull*), e que pode ser paralelizado com maior facilidade.

1.1.7 Diagrama de Voronoi, triangulação de Delaunay e problemas de proximidade

Esta seção vai analisar uma série de problemas correlacionados e que envolvem o conceito de proximidade no plano. O principal recurso de que se dispõe para resolver este tipo de problema é o *diagrama de Voronoi*, uma estrutura geométrica proposta no início do século e que é capaz de responder uma grande variedade de perguntas a respeito de proximidade em um conjunto de pontos: qual ponto está mais próximo, qual o mais distante, entre outras. Diversos trabalhos buscaram destacar a importância desta estrutura para SIG [Gold91][Gold92b][OBS92], mas até o momento os pacotes comerciais não a incorporam explicitamente, para prejuízo de aplicações que demandam com freqüência a solução de problemas de proximidade.

1.1.7.1 Diagramas de Voronoi

Seja $P = \{p_1, p_2, \ldots, p_n\}$ um conjunto de pontos no plano, usualmente denominados *locais* (ou *sites*). O plano pode ser particionado de modo que cada ponto esteja associado ao elemento de *P* mais próximo de si. O conjunto dos pontos associados ao local p_i constituem o *polígono de Voronoi*[16] de p_i, denotada $V(p_i)$. Este polígono é, portanto, o lugar geométrico dos pontos do plano mais próximos de p_i do que de qualquer outro elemento de *P*, ou seja

$$V(p_i) = \{x \mid dist(p_i - x) \leq dist(p_j - x), \forall j \neq i\}$$

Existe a possibilidade de se ter pontos que são igualmente próximos a dois ou mais locais. O conjunto destes pontos constitui o *diagrama de Voronoi* para o conjunto de locais, denotado *Vor(P)*.

A construção do diagrama pode ser melhor compreendida observando o que ocorre quando o número de locais vai sendo aumentado gradativamente. Inicialmente, considere-se apenas dois locais, p_1 e p_2. O diagrama de Voronoi consiste na reta que secciona ao meio o segmento p_1p_2 e é perpendicular a este, a *mediatriz* do segmento (Figura 1.55). Todos os pontos da reta são igualmente próximos a p_1 e a p_2. Pontos no semiplano que contém p_1 constituem o polígono de Voronoi correspondente a p_1, e analogamente o outro semiplano corresponde a $V(p_2)$.

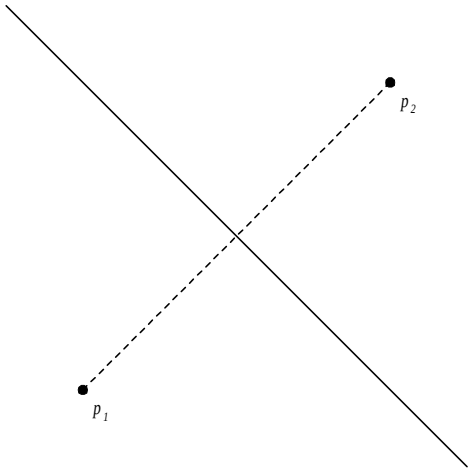


Figura 1.55 - Diagrama de Voronoi com dois locais

Expandindo para três locais, é fácil perceber que o diagrama de Voronoi será formado pelas semi-retas que cortam as arestas de $p_1p_2p_3$ ao meio e segundo uma perpendicular, portanto as mediatrizes das arestas, partindo do *circuncentro* do triângulo (Figura 1.56). O circuncentro é o centro do círculo definido pelos vértices do triângulo, e é possível que ele não pertença ao triângulo.

---

[16] Estes polígonos recebem diversos nomes alternativos na literatura: regiões de Dirichlet, polígonos de Thiessen, células de Wigner-Seitz, ou regiões proximais [PrSh88]

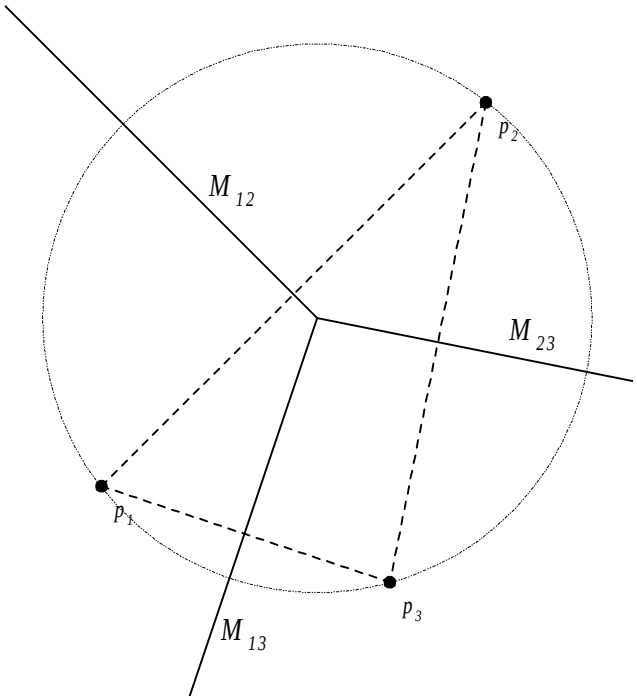


Figura 1.56 - Diagrama de Voronoi com três locais

Generalizando para um número maior de locais, fica claro que o processo de construção deve levar em conta as mediatrizes dos segmentos definidos entre cada par de locais. A mediatriz entre os locais p_i e p_j será denotada como M_{ij}. Seja S_{ij} o semiplano definido por M_{ij} e que contém p_i. Então S_{ij} contém todos os pontos do plano que estão mais próximos de p_i do que de p_j. Para obter o polígono de Voronoi de p_i, é necessário combinar todos os semiplanos S_{ij} com $i \neq j$, e portanto

$$V(p_i) = \bigcap_{i \neq j} S_{ij}$$

Como semiplanos são, por definição, convexos (não existe nenhum segmento definido entre dois pontos do semiplano e que contém pontos que não pertençam a ele). A interseção de conjuntos convexos é também um conjunto convexo [PrSh88]. Portanto, pode-se concluir que qualquer polígono de Voronoi é convexo também.

A Figura 1.57 mostra um exemplo de diagrama de Voronoi com 12 locais. Observe-se que existem 12 polígonos, um para cada local. Os vértices dos polígonos estão ligados, em geral, a três arestas, exceto quando se tem quatro ou mais locais co-circulares em *P* (Figura 1.58a) Esta situação é considerada degenerada, mas pode ser eliminada com a introdução de perturbações infinitesimais nos pontos co-circulares (Figura 1.58b).

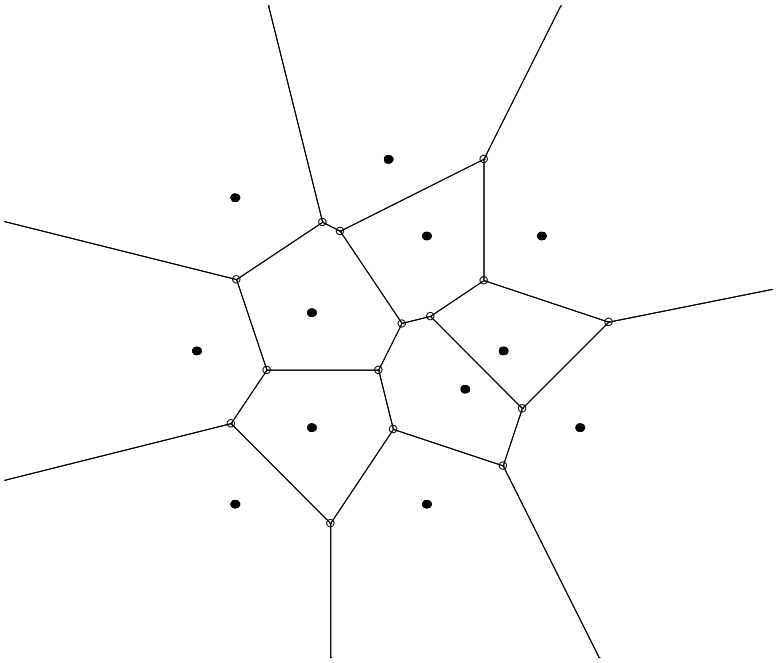


Figura 1.57 - Diagrama de Voronoi com 12 locais

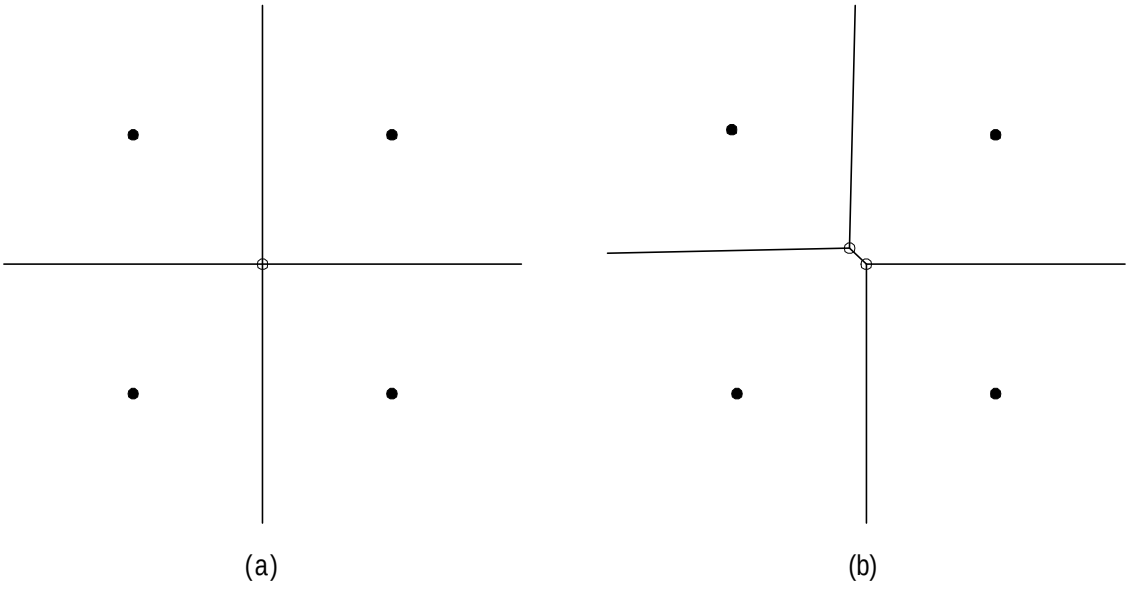


Figura 1.58 - Quatro pontos co-circulares e perturbação

1.1.7.2 Triangulação de Delaunay

Assim, considerando que o máximo grau de um vértice seja igual a três, define-se o dual de um diagrama de Voronoi através de um grafo *G*, cujos nós correspondem aos locais e cujos arcos conectam locais cujos polígonos de Voronoi compartilham uma aresta do diagrama. Demonstra-se que este grafo é *planar*, e portanto vale a fórmula de Euler: em um grafo planar com n vértices, existem $3n-6$ arcos e $2n-4$ faces [ORou94].

Traçando *G* com linhas retas entre os vértices (locais), assumindo que não existam quatro vértices co-circulares, é produzida uma *triangulação* de *P*, que é denominada

triangulação de Delaunay e denotada *D(P)*. A Figura 1.59 mostra a triangulação de Delaunay sobre o diagrama de Voronoi da Figura 1.57.

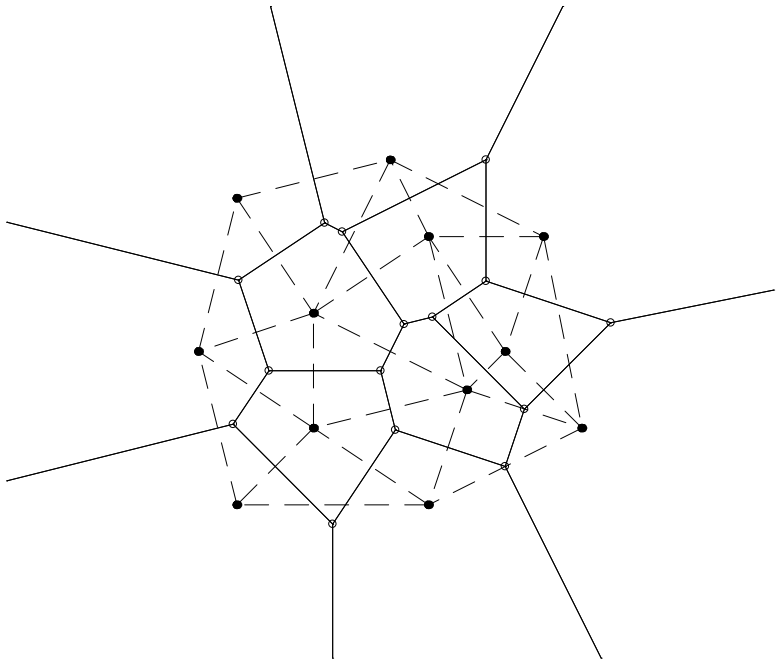


Figura 1.59 - Diagrama de Voronoi (linhas cheias) e triangulação de Delaunay (linhas pontilhadas)

[ORou94] e [PrSh88] listam as propriedades da triangulação de Delaunay e dos diagramas de Voronoi. As mais interessantes são as seguintes:

- A cada triângulo de *D(P)* corresponde um vértice de *Vor(P)*;
- A cada nó de *D(P)* corresponde um polígono de *Vor(P)*;
- A fronteira de *D(P)* coincide com o fecho convexo dos locais de *P*;
- O interior de cada triângulo de *D(P)* não contém nenhum local;
- $V(p_i)$ é ilimitado se e somente se p_i pertence ao fecho convexo de *P*;
- Se *v* é um vértice do diagrama de Voronoi, na junção de $V(p_1)$, $V(p_2)$ e $V(p_3)$, então *v* é o centro do círculo definido por p_1, p_2 e p_3;
- Se p_j é o vizinho mais próximo de p_i, então p_ip_j é uma aresta de *D(P)*.

1.1.7.3 Implementação

Devido à extensão do código fonte, este capítulo não irá detalhar a implementação de algoritmos para obter o diagrama de Voronoi e a triangulação de Delaunay. As

principais idéias e alternativas apresentadas na literatura de geometria computacional, no entanto, serão delineadas, deixando a cargo do leitor um aprofundamento maior[17].

A construção do diagrama de Voronoi pode ser implementada usando a técnica de dividir para conquistar, gerando um algoritmo ótimo $O(n \log n)$ [PrSh88]. Em linhas gerais, o algoritmo consiste em:

- particionar o conjunto inicial de pontos P em dois subconjuntos, P_1 e P_2, de tamanho aproximadamente igual;
- construir $Vor(P_1)$ e $Vor(P_2)$ recursivamente;
- combinar $Vor(P_1)$ e $Vor(P_2)$ para obter $Vor(P)$.

A maior dificuldade desta alternativa está em conseguir implementar a etapa de combinação de dois diagramas em tempo $O(n)$ para que o custo total não exceda $O(n \log n)$. [PrSh88] apresenta uma descrição detalhada deste procedimento, que é relativamente complexo e de implementação difícil.

Uma alternativa a esta forma de implementação foi desenvolvida por Fortune ([Fort87] *apud* [ORou94]), usando a técnica de varredura do plano. Foi necessário conceber uma variação da varredura tradicional, pois na construção do diagrama a linha de varredura vai certamente interceptar o polígono de Voronoi bem antes de encontrar o local correspondente. Assim, Fortune propôs a construção de cones invertidos, com ápice colocado sobre cada local e com lados cuja inclinação é de 45 graus. A projeção da interseção entre os cones sobre o plano XY é exatamente o diagrama de Voronoi. Para recuperar o diagrama, o algoritmo faz a varredura do conjunto de cones utilizando um plano inclinado também a 45 graus em relação ao plano XY. A interseção do plano com cada cone configura uma parábola, e o algoritmo usa uma composição destas parábolas (denominada "frente parabólica") no lugar da linha de varredura tradicional. A frente parabólica consegue, portanto, detectar arestas do diagrama antes de passar pelos locais propriamente ditos. O algoritmo final tem também complexidade $O(n \log n)$.

Outra linha de raciocínio defende a construção da triangulação de Delaunay como passo intermediário para a obtenção do diagrama de Voronoi, já que foi demonstrado [FiCa91] que a triangulação de Delaunay pode ser transformada no diagrama de Voronoi em tempo linear, e vice-versa. Um algoritmo usando a técnica de dividir para conquistar, muito semelhante ao apresentado para o diagrama de Voronoi, foi proposto por Lee e Schachter ([LeSc80] *apud* [FiCa91]). Também neste algoritmo o desafio maior é a etapa de combinação de triangulações parciais em tempo linear. Para facilitar a construção da triangulação é utilizada uma estrutura topológica *winged-edge* (vide seção 1.2.2). Esta

---

[17] Existem diversos programas para a construção do diagrama de Voronoi e da triangulação de Delaunay disponíveis na Internet. O URL `http://www.geog.umn.edu/software/cglist` contém *links* para diversas páginas que oferecem *freeware* na área de geometria computacional. O programa original de Fortune, por exemplo, pode ser encontrado em `http://netlib.bell-labs.com/netlib/voronoi/index/html`, incluindo o código fonte. O mesmo programa é capaz de produzir o diagrama de Voronoi, a triangulação de Delaunay e ainda o fecho convexo de um conjunto de pontos.

proposta foi posteriormente refinada por Guibas e Stolfi [GuSt85], utilizando uma nova estrutura topológica denominada *quad-edge* (vide seção 1.2.3) Um aperfeiçoamento deste algoritmo foi proposto posteriormente, obtendo significativos ganhos de desempenho e também uma maior facilidade de implementação [Leac92].

1.1.7.4 Aplicações

O diagrama de Voronoi tem muitas aplicações em SIG. Como estrutura de dados básica para resolver problemas de proximidade, pode ser empregada sempre que o número de consultas justificar o custo de sua criação e manutenção. Apesar das vantagens que se tem em usar esta estrutura na solução de problemas como os relacionados a seguir, a maioria das aplicações baseadas em SIG comerciais prefere adotar estratégias mais genéricas, usando os recursos básicos de localização individual de objetos disponíveis no SIG subjacente. No entanto, a grande utilidade do diagrama de Voronoi no contexto de SIG tem conduzido a propostas no sentido de incrementar seu uso, incorporando estratégias de manutenção dinâmica e a possibilidade de trabalhar com objetos móveis em tempo real [Gold92b][OBS92].

O mesmo ocorre com a triangulação de Delaunay, usada sempre que se tem a necessidade de particionar o plano com base em um conjunto de pontos. São freqüentes as situações em que os pontos representam locais onde se consegue amostrar alguma variável física, como altitude, temperatura ou índice de chuvas, sendo necessário produzir um mapeamento contínuo da variável para toda uma região de interesse. Nestas situações, a triangulação de Delaunay é imbatível, pois gera um resultado em que é maximizado o menor ângulo interno de cada triângulo [Edel87], o que confere à malha triangular uma aparência mais regular.

A seguir, são apresentadas algumas aplicações do diagrama de Voronoi e da triangulação de Delaunay que são significativas no contexto de SIG, sem a pretensão de esgotar o assunto.

Ponto mais próximo. Dado um conjunto P de locais, deseja-se saber qual é o mais próximo de um ponto q dado. Trata-se de uma consulta tradicional em geoprocessamento, e tem diversas aplicações em análise espacial. Quando se dispõe do diagrama de Voronoi, a solução do problema consiste simplesmente em determinar o local correspondente ao polígono de Voronoi que contém q. Naturalmente, existem maneiras de resolver este problema sem utilizar o diagrama de Voronoi (por exemplo, analisar seqüencialmente a distância entre q e todos os pontos $p_i \in P$), mas quando se pretende realizar um número razoável de consultas, sobre um conjunto de pontos relativamente estável, construir antecipadamente o diagrama de Voronoi pode ser muito vantajoso. Um exemplo prático: informar aos usuários do sistema de transporte coletivo qual é o ponto de parada de ônibus mais próximo de sua residência ou local de trabalho. Outro exemplo: localizar o hidrante mais próximo de um determinado prédio.

Foi demonstrado que, contando com alguma estrutura de indexação espacial, problemas como este (denominados genericamente de *point location problems*: determinar em qual partição do plano está um determinado ponto) podem ser resolvidos em tempo $O(\log n)$.

Uma variação deste problema consiste em localizar os k pontos de P mais próximos de q. Isto é possível usando o chamado *diagrama de Voronoi de ordem k*. Conforme foi

apresentado, o diagrama de Voronoi tradicional é o diagrama de ordem 1, ou seja, cada polígono é o lugar geométrico dos pontos do plano mais próximos do local p_i do que de qualquer outro. Analogamente, um diagrama de Voronoi de ordem k é formado por polígonos que constituem o lugar geométrico dos pontos do plano cujos k vizinhos mais próximos são os locais p_{i1}, p_{i2}, p_{i3},...p_{ik}, em qualquer ordem. Algoritmos para obter tais diagramas têm complexidade $O(n^3)$, e não serão discutidos aqui (para maior aprofundamento, vide [PrSh88], [Edel87] e [ORou94]).

Vizinhos mais próximos. Este problema consiste em, dado um conjunto P de locais, encontrar o local p_j mais próximo de cada local p_i (o vizinho mais próximo de p_i) pertencente a P. Um exemplo prático está na geração de indicações de "quiosque mais próximo" para caixas automáticos, de modo a informar os clientes sobre alternativas em caso de falha de algum deles. Outro exemplo é, dado um conjunto de aeroportos satisfazendo a certas condições (comprimento da pista, instrumentação de pouso, equipamentos de emergência), determinar qual é o aeroporto mais indicado para redirecionamento dos pousos em caso de fechamento por mau tempo de qualquer um dos aeroportos.

O resultado do problema é um *grafo de proximidade* (Figura 1.60), definido como sendo um grafo em que os nós são os locais e as arestas são direcionadas entre cada local e seu vizinho mais próximo. Este grafo pode ser usado para responder com facilidade a questões como "qual é o local mais próximo deste" e "que locais são mais próximos deste do que de qualquer outro". No segundo exemplo, estas questões seriam traduzidas como "qual é o aeroporto mais próximo do Aeroporto de Confins" e "que aeroportos redirecionarão seus pousos para Confins em caso de fechamento". Observe que este grafo não é necessariamente conexo, e também que as ligações entre os nós não são sempre reflexas.

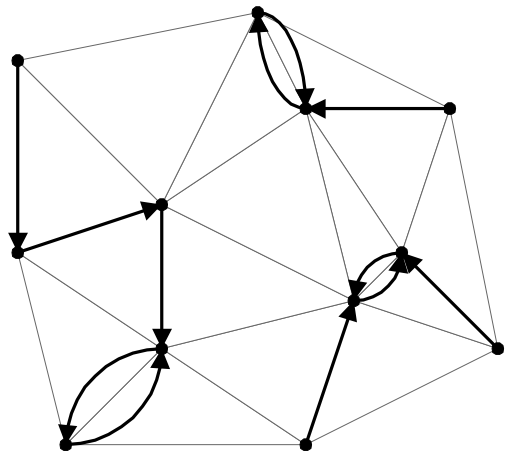


Figura 1.60 - Grafo de proximidade sobre a triangulação de Delaunay da Figura 1.59

Este problema pode ser resolvido em tempo linear a partir da disponibilidade do diagrama de Voronoi. Basta analisar cada aresta definida no diagrama, calcular a distância entre os locais separados por aquela aresta, e manter os menores valores em associação com cada local. Como o número de arestas do diagrama é, no máximo, igual a $3n - 6$ (vide seção 1.1.7.2), o tempo necessário para esta análise é $O(n)$. Naturalmente,

esta tarefa será facilitada pela construção de uma estrutura de dados adequada ao problema, como a *winged-edge* (vide seção 1.2.2). Uma alternativa a esta solução é utilizar a triangulação de Delaunay, verificando para cada vértice de triângulo (correspondente a um local) qual é a aresta de menor comprimento que incide sobre ele. O local na outra extremidade desta aresta é o vizinho mais próximo. A validade desta estratégia decorre do fato de que o grafo de proximidade está contido em $D(P)$ [ORou94].

Maior círculo vazio. Consiste em determinar o maior círculo cujo centro pertence ao fecho convexo dos locais de P e que não contém nenhum local. Foi demonstrado que, caso o centro do círculo seja interior ao fecho convexo, então deve coincidir com um vértice de Voronoi. O centro do maior círculo vazio pode também estar sobre alguma aresta do fecho convexo, e neste caso estará situado no ponto médio da aresta, eqüidistante a dois vértices [ORou94]. Um algoritmo para resolver este problema a partir do diagrama de Voronoi consiste, portanto, em verificar o raio do círculo definido sobre cada vértice, que corresponde à distância do vértice a um dos três locais eqüidistantes a ele. Também deve ser verificado o raio de cada círculo definido com centro no ponto médio de cada aresta do fecho convexo. Ao final, o resultado é o círculo de maior raio dentre os verificados (Figura 1.61).

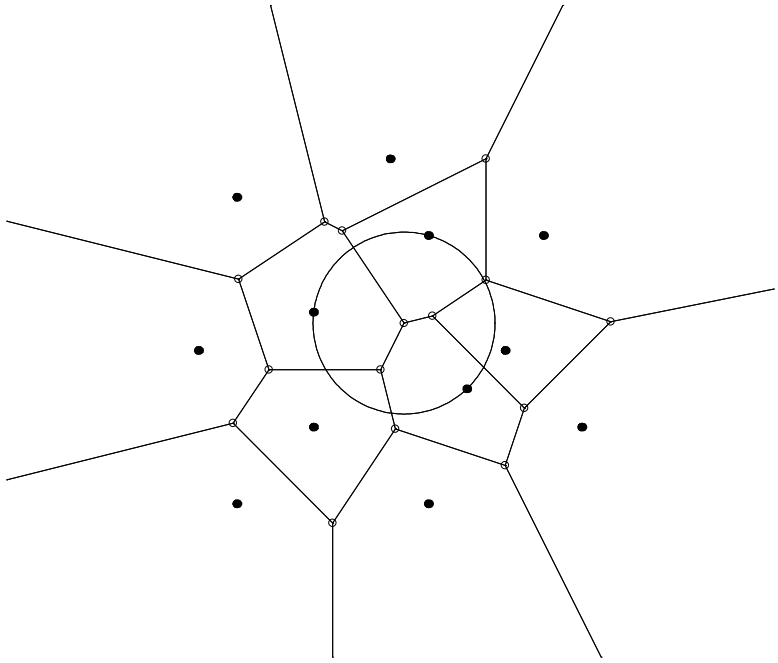


Figura 1.61 - Maior círculo vazio

Esta técnica pode ser usada para escolher o ponto de implantação de um novo empreendimento, desde que se faça algumas hipóteses simplificadoras: (1) a distribuição da clientela pelo território é uniforme, e (2) é possível encontrar espaço para a implantação do empreendimento no centro aproximado do círculo vazio. Caso estas simplificações não sejam aceitáveis, torna-se necessário utilizar técnicas mais avançadas para escolher o local do empreendimento, tais como a determinação da *superfície de potencial de vendas*, ou semelhante.

Outra situação em que este problema se aplica está na escolha de uma área para a localização de uma atividade potencialmente poluidora, em um lugar que seja tão distante quanto possível de centros populacionais.

Modelagem Digital de Terrenos. A triangulação de Delaunay é muito usada na criação de modelos digitais do terreno (MDT). Na criação de MDT, parte-se em geral de um conjunto de amostras (pontos cotados) distribuídos de maneira irregular pelo terreno. As fontes de informação são a restituição de fotos aéreas por processo estereoscópico ou levantamentos topográficos de campo. A partir destas amostras deseja-se construir uma superfície que represente o relevo do local. Para isso, é criada uma triangulação contida no fecho convexo dos pontos amostrais, e a superfície é então aproximada pelos triângulos tridimensionais formados. Como os três vértices de um triângulo definem um plano, esta estratégia eqüivale a imaginar que o relevo varia linearmente entre dois pontos cotados conhecidos, o que é suficiente para a maioria das aplicações. Se o grau de aproximação obtido não for satisfatório, pode-se densificar a malha de triângulos, introduzindo novos pontos.

A partir da triangulação (denominada por alguns SIG de TIN - *Triangulated Irregular Network*) pode-se produzir mapas de curvas de nível, usando um algoritmo simples. Basta determinar a interseção de cada triângulo com o plano correspondente à cota da curva de nível que se pretende traçar . Esta interseção pode ser nula (Figura 1.62a), pode ser um ponto (Figura 1.62b), um segmento de reta (Figura 1.62c) ou mesmo todo o triângulo (Figura 1.62d) [Bour87]. Nos casos em que a interseção é um segmento, este é determinado e armazenado para compor a curva de nível. Os outros casos podem ser ignorados para efeito de traçado das curvas. A continuidade de cada curva está garantida pelo fato de que os pontos extremos do segmento de interseção entre plano e triângulo são os mesmos nos triângulos adjacentes.

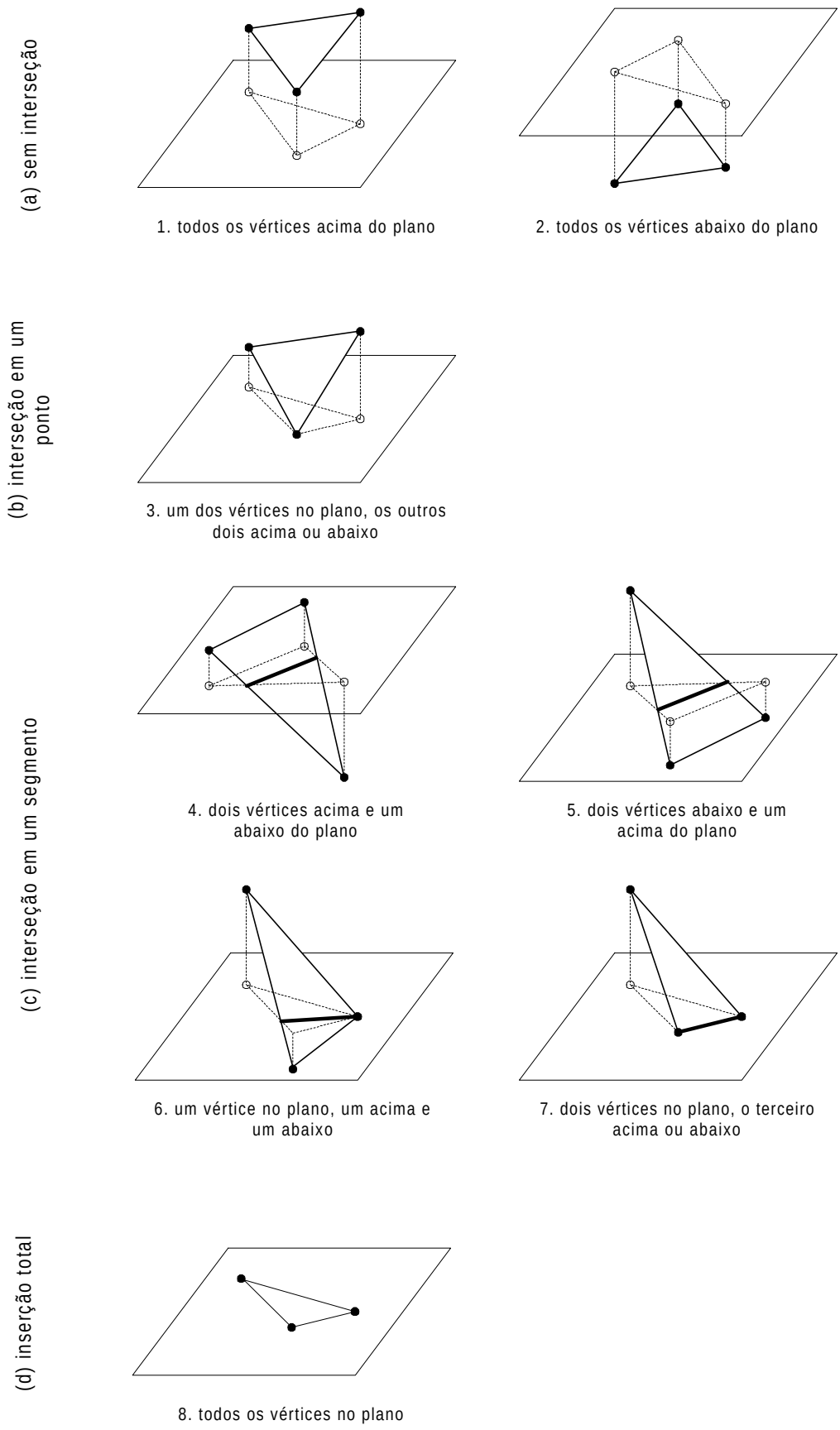


Figura 1.62 - Interseção entre triângulo e plano

Naturalmente, a qualidade do resultado depende da densidade de triângulos – quanto mais triângulos, mais refinado será o aspecto da curva gerada, especialmente nas regiões onde o relevo muda mais bruscamente. Mesmo assim, curvas de nível traçadas por este processo tendem a adquirir um aspecto anguloso, pouco natural. Assim, é executada uma etapa de pós-processamento, em que a curva é suavizada por uma *spline* ou outro processo qualquer. Observe-se que é necessário que a suavização não altere as características topológicas das curvas de nível, ou seja, as curvas suavizadas, assim como as originais, não podem se interceptar.

Um tipo de consulta freqüentemente associado com o MDT é a determinação da cota do terreno em um ponto *q* dado. Isto pode ser feito determinando qual triângulo contém *q*, e fazendo uma interpolação linear entre os vértices do triângulo para obter a cota procurada. Existem também processos de interpolação que utilizam as curvas de nível para obter a cota de pontos intermediários quaisquer.

O mesmo processo descrito para MDT pode ser usado para tratar outras variáveis de distribuição contínua e que são medidas em pontos discretos, tais como temperatura, índice pluviométrico ou ruído ambiental.

1.2 Topologia

1.2.1 Conceitos básicos (capítulo da apostila)

1.2.2 A estrutura Winged-Edge

1.2.3 Outras estruturas topológicas - Quad-edge?

1.2.4 Topologia de redes

1.2.5 Aplicações

Pesquisas de proximidade em diagramas de Voronoi codificados segundo uma winged-edge!

OBS: Incluir discussão sobre o uso de estruturas topológicas em SIG: custo de manutenção x custo de geração, considerando a freqüência de uso

1.3 Indexação Espacial

Estruturas reativas - Oosterom

1.4 Referências

[Baas88] Baase, S. *Computer Algorithms: Introduction to Design and Analysis*, 2nd Edition, Addison-Wesley, 1988.

[BCA95] Barber, C., Cromley, R., and Andrle, R. Evaluating Alternative Line Simplification Strategies for Multiple Representations of Cartographic Lines. *Cartography and Geographic Information Systems* 22(4): 276-290, 1995.

[Bear91] Beard, K. Theory of the Cartographic Line Revisited: Implications for Automated Generalization. *Cartographica* 28(4): 32-58, 1991.

[Bour87] Bourke, P. D. A Contouring Subroutine. *BYTE* June 1987, 143-150.

[Butt85] Buttenfield, B. P. Treatment of the Cartographic Line. *Cartographica* 22(2): 1-26, 1985.

[Butt89] Buttenfield, B. P. Scale Dependence and Self-Similarity in Cartographic Lines. *Cartographica* 26(1): 79-100, 1989.

[ChCh96] Chan, W. S. and Chin, F. Approximation of Polygonal Curves with Minimum Number of Line Segments or Minimum Error. *International Journal of Computational Geometry and Applications* 6(1): 59-77, 1996.

[CLR90] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*. McGraw-Hill and MIT Press, 1990.

[CrCa91] Cromley, R. G., and Campbell, G. M. Noninferior Bandwidth Line Simplification: Algorithm and Structural Analysis. *Geographical Analysis* 23 (1): 25-38, 1991.

[CrCa92] Cromley, R. G. and Campbell, G. M. Integrating Quantitative and Qualitative Aspects of Digital Line Simplification. *The Cartographic Journal* 29(1): 25-30, 1992.

[Crom88] Cromley, R. G. A Vertex Substitution Approach to Numerical Line Simplification. In *Proceedings of the Third International Symposium on Spatial Data Handling*, 57-64, 1988.

[Crom91] Cromley, R. G. Hierarchical Methods of Line Simplification. *Cartography and Geographic Information Systems* 18(2): 125-131, 1991.

[Davi97] Davis Jr., C. A. Uso de Vetores em GIS. *Fator GIS* 4(21): 22-23, 1997.

[Deve85] Deveau, T. J. Reducing the number of points in a plane curve representation. In *Proceedings of AutoCarto 7*, 152-160, 1985.

[DGHS88] Dobkin, D., Guibas, L., Hershberger, J., and Snoeyink, J. An Efficient Algorithm for Finding the CSG Representation of a Simple Polygon.

Computer Graphics 22(4):31-40, 1988.

[DoPe73] Douglas, D. H. and Peucker, T. K. Algorithms for the Reduction of the Number of Points Required to Represent a Line or its Caricature. *The Canadian Cartographer* 10(2): 112-122, 1973.

[DuHa73] Duda, R. and Hart, P. *Pattern Classification and Scene Analysis.* John Wiley & Sons, New York, 1973.

[Edel87] Edelsbrunner, H. *Algorithms in Combinatorial Geometry*, Springer-Verlag, 1987.

[FiCa91] Figueiredo, L. H., Carvalho, P. C. P. *Introdução à Geometria Computacional*, Instituto de Matemática Pura e Aplicada, 1991.

[Fort87] Fortune, S. A Sweepline Algorithm for Voronoi Diagrams. *Algorithms* 2:153-174, 1987.

[GHMS93] Guibas, L. J., Hershberger, J. E., Mitchell, J. S. B., and Snoeyink, J. S. Approximating Polygons and Subdivisions with Minimum Link Paths. *International Journal of Computational Geometry and Applications* 3(4): 383-415, 1993.

[Gold91] Gold, C. M. Problems with Handling Spatial Data - The Voronoi Approach. *CISM Journal* 45: 65-80, 1991.

[Gold92b] Gold, C. M. Dynamic Spatial Data Structures - The Voronoi Approach. In *Proceedings of, the Canadian Conference on GIS*, 245-255, 1992.

[GuSt85] Guibas, L. and Stolfi, J. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Transactions on Graphics* 4(2): 74-123, 1985.

[HeSn92] Hershberger, J. and Snoeyink, J. Speeding Up the Douglas-Peucker Line-Simplification Algorithm. In *Proceedings of the Fifth International Symposium on Spatial Data Handling*, 1: 134-143, 1992.

[Horn85] van Horn, E. K. Generalizing Cartographic Databases. In *Proceedings of AutoCarto 7*, 532-540, 1985.

[ImIr86] Imai, H. and Iri, M. Polygonal Approximations of a Curve – Formulations and Algorithms. In Toussaint, G. T. (editor) *Computational Morphology*, North Holland, 1988.

[Jain89] Jain, A. K. *Fundamentals of Digital Image Processing*. Prentice-Hall, 1989.

[Jenk81] Jenks, G. F. Lines, Computers and Human Frailties. In *Annals of the Association of American Geographers* 71(1): 1-10, 1981.

[Jenk89] Jenks, G. F. Geographic Logic in Line Generalization. *Cartographica*

26(1): 27-42, 1989.

[Joha74] Johannsen, T. M. A Program for Editing and for Some Generalizing Operations (For Derivation of a Small Scale Map from Digitized Data in 1:50,000). In Csati, E. (editor) *Automation: The New Trend in Cartography*, The Geocartotechnical Research Department (Budapest, Hungary), 131-138, 1974.

[Knut73] Knuth, D. E. *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Edition. Addison-Wesley, 1973.

[Knut73a] Knuth, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 2nd Edition, Addison-Wesley, 1973.

[Lang69] Lang, T. Rules for Robot Draughtsmen. *Geographical Magazine* 22: 50-51, 1969.

[LaTh92] Laurini, R. and Thompson, D. *Fundamentals of Spatial Information Systems*. Academic Press, 1992.

[Leac92] Leach, G. Improving Worst-Case Optimal Delaunay Triangulation Algorithms. In *Proceedings of the Fourth Canadian Conference on Computational Geometry*, 1992.

[LeSc80] Lee, D. T., Schachter, B. J. Two Algorithms for Constructing the Delaunay Triangulation. *International Journal of Computer and Information Science* 9(3):219-242, 1980.

[LiOp92] Li, Z. and Openshaw, S. Algorithms for Automated Line Generalization Based on a Natural Principle of Objective Generalization. *International Journal of Geographic Information Systems* 6(5): 373-389, 1992.

[MaKn89] Margalit, A., Knott, G. D. An Algorithm for Computing the Union, Intersection or Difference of Two Polygons. *Computers & Graphics* 13(2): 167-183, 1989.

[Mari79] Marino, J. S. Identification of Characteristic Points Along Naturally Occurring Lines: An Empirical Study. *The Canadian Cartographer* 16:70-80, 1979.

[McMa86] McMaster, R. B. A Statistical Analysis of Mathematical Measures for Linear Simplification. *American Cartographer* 13:103-116, 1986.

[McMa87a] McMaster, R. B. Automated Line Generalization. *Cartographica* 24(2):74-111, 1987.

[McMa87b] McMaster, R. B. The Geometric Properties of Numerical Generalization. *Geographical Analysis* 19(4): 330-346, 1987.

[McMa89] McMaster, R. B. The Integration of Simplification and Smoothing Algorithms in Line Generalization. *Cartographica* 26(1): 101-121,

1989.

[McSh92] McMaster, R. B. and Shea, K. S. *Generalization in Digital Cartography*. Association of American Geographers, 1992.

[Melk87] Melkman, A. A. On-line Construction of the Convex Hull of a Simple Polyline. *Information Processing Letters* 25:11-12, 1987.

[Melk88] Melkman, A. A. and O'Rourke, J. On Polygonal Chain Approximation. In Toussaint, G. T. (editor) *Computational Morphology*, North Holland, 1988.

[Mowe96] Mower, J. Developing Parallel Procedures for Line Simplification. *International Journal of Geographical Information Science* 10 (6): 699-712, 1996.

[Mull87] Muller, J. C. Optimum Point Density and Compaction Rates for the Representation of Geographic Lines. In *Proceedings of AutoCarto 8*, 221-230, 1987.

[Mull90a] Muller, J. C. The Removal of Spatial Conflicts in Line Generalization. *Cartography and Geographic Information Systems* 17 (2): 141-149, 1990.

[Mulm94] Mulmuley, K. *Computational Geometry: An Introduction Through Randomized Algorithms*, Prentice-Hall, 1994.

[Nick88] Nickerson, B. G. Automated Cartographic Generalization for Linear Features. *Cartographica* 25 (3): 15-66, 1988.

[NiPr82] Nievergelt, J., Preparata, F. P. Plane-Sweep Algorithms for Intersecting Geometric Figures. *Communications of the ACM* 25(10): 739-747, 1982.

[OBS92] Okabe, A., Boots, B., Sugihara, K. *Spatial Tesselations: Concepts and Applications of Voronoi Diagrams*, John Wiley, 1992.

[OoSc95] van Oosterom, P. and Schenkelaars, V. The Development of an Interactive Multi-Scale GIS. *International Journal of Geographical Information Systems* 9(5), 489-507, 1995.

[Oost93] van Oosterom, P. *Reactive Data Structures for Geographic Information Systems*, Oxford University Press, 1993.

[Ophe81] Opheim, H. Smoothing a Digitized Curve by Data Reduction Methods. In Encarnação, J. L. (editor) *Proceedings of Eurographics '81*, 127-135, North-Holland, 1981.

[ORou94] O'Rourke, J. *Computational Geometry in C*, Cambridge University Press, 1994.

[PAF95] Plazanet, C., Affholder, J.-G., and Fritsch, E. The Importance of Geometric Modeling in Linear Feature Generalization. *Cartography and Geographic Information Systems* 22(4): 291-305, 1995.

[Perk66] Perkal, J. An Attempt at Objective Generalization. *Michigan Inter-University Community of Mathematical Geographers*, Discussion Paper No. 10, 1966.

[Peuc75] Peucker, T. K. A Theory of the Cartographic Line. *International Yearbook of Cartography* 16: 134-143, 1975.

[Plaz95] Plazanet, C. Measurements, Characterization and Classification for Automated Line Feature Generalization. In *Proceedings of AutoCarto 12*, 59-68, 1995.

[PrSh88] Preparata, F. P., Shamos, M. I. *Computational Geometry: an Introduction*, Springer-Verlag, 1988.

[Rame72] Ramer, U. An Iterative Procedure for the Polygonal Approximation of Plane Curves. *Computer Vision, Graphics, and Image Processing* 1:244-256, 1972.

[ReWi74] Reumann, K. and Witkam, A. P. M. Optimizing Curve Segmentation in Computer Graphics. In *Proceedings of the International Computing Symposium 1973*, 467-472, North-Holland, 1974.

[Robe85] Roberge, J. A Data Reduction Algorithm for Planar Curves. *Computer Vision, Graphics, and Image Processing* 1(3): 244-256, 1985.

[RSM78] Robinson, A. H., Sale, R. D. and Morrison, J. L. *Elements of Cartography*, 4th Edition, John Wiley & Sons, 1978.

[Schn97] Schneider, M. *Spatial Data Types for Database Systems: Finite Resolution Geometry for Geographic Information Systems*, Lecture Notes in Computer Science 1288. Springer-Verlag, 1997.

[Sedg90] Sedgewick, R. *Algorithms in C*, Addison-Wesley, 1990.

[Tarj83] Tarjan, R. E. *Data Structures and Network Algorithms*, Regional Conference Series in Applied Mathematics 44, SIAM, 1983.

[Thap88] Thapa, K. Automatic Line Generalization using Zero Crossings. *Photogrammetric Engineering and Remote Sensing* 54(4), 511-517, 1988.

[Tobl64] Tobler, W. R. An Experiment in the Computer Generalization of Maps. Technical Report No. 1, Office of Naval Research Task No. 389-137, Contract No. 1224 (48), Office of Naval Research, Geography Branch, 1964.

[ToPi66] Topfer, F. and Pillewizer, W. The Principles of Selection. *Cartographic Journal* 3 (10-16), 1966.

[Tous83] Toussaint, G. T. Computing Largest Empty Circles with Location Constraints. *International Journal of Computer and Information Science* 12(5): 347-358, 1983.

[Vatt92] Vatti, B. R. A Generic Solution to Polygon Clipping. *Communications of the ACM* 35(7): 57-63, 1992.

[ViWh90] Visvalingam, M. and Whyatt, J. The Douglas-Peucker Algorithm for Line Simplification: Re-evaluation through Visualization. *Computer Graphics Forum* 9 (213-228), 1990.

[ViWh93] Visvalingam, M. and Whyatt, J. D. Line Generalisation by Repeated Elimination of Points. *Cartographic Journal* 30 (1): 46-51, 1993.

[ViWi95] Visvalingam, M. and Williamson, P. J. Simplification and Generalization of Large Scale Data for Roads: A Comparison of Two Filtering Algorithms. *Cartography and Geographic Information Systems* 22 (4), 264-275, 1995.

[Weib95] Weibel, R. Map Generalization in the Context of Digital Systems. *Cartography and Geographic Information Systems*, Guest Editorial to Special Issue on Automated Map Generalization, 22(4): 3-10, 1995.

[Whit85] White, E. R. Assessment of Line-Generalization Algorithms Using Characteristic Points. *The American Cartographer* 12(1): 17-28, 1985.

[Will78] Williams, C. M. An Efficient Algorithm for the Piecewise Linear Approximation of a Polygonal Curve in the Plane. *Computer Vision, Graphics, and Image Processing* 8:286-293, 1978.

[Wolb90] Wolberg, G. *Digital Image Warping*. IEEE Computer Society Press, 1990.

[ZhSa97] Zhao, Z. and Saalfeld, A. Linear-Time Sleeve-Fitting Polyline Simplification Algorithms. In *Proceedings of AutoCarto 13*, 1997.

[Zivi96] Ziviani, N. *Projeto de Algoritmos com Implementações em Pascal e em C*, 3ª. Edição, Editora Pioneira, 1996.