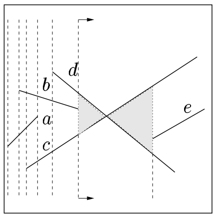
MAC 5747 GEOMETRIA COMPUTACIONAL

INTERSECÇÃO DE SEGMENTOS

1. Introdução



Um dos problemas mais básicos em geometria computacional é o de computar intersecção. O cálculo de intersecções no plano e no espaço 3-dimensional é uma operação básica em diversas áreas: em robótica e planejamento de movimento é importante sabermos quando dois objetos se intersectam para evitarmos colisões; em computação gráfica, *ray shooting* é um método importante para a 'renderização' de cenas, a parte que é computacionalmente mais 'cara' do ray shooting é justamente determinar intersecções entre o raio com outros objetos.

Muitos problemas complexos de intersecção são decompostos a problemas de intersecção mais simples. Estudaremos a seguir algoritmos básicos de intersecção, sobre os quais alguns algoritmos complexos se apoiam.

O problema que consideraremos é: *dados:* n segmentos de retas no plano; *determinar:* todos os pontos de intersecção entre pares de segmentos. Outro problema relacionado com este é o Problema do Teste de Intersecção entre Segmentos (*Line-Segment Intersection Test*) que consiste em: *dados* n segmentos de reta no plano; *decidir:* se existem dois segmentos que se intersectam. O primeiro problema pede todas as inrtersecções e o segundo se satisfaz com uma resposta SIM ou NÃO.

Observe que n segmentos podem se intersectar em $\Omega(n^2)$ pontos distintos (veja Figura 1). Assim poderíamos colocar como nosso objetivo desenvolver um algoritmo de complexidade de tempo $O(n^2)$ e afirmar que a complexidade de tempo assintótica desse algoritmo é ótima. Na realidade, o algoritmo ingênuo óbvio para o problema tem complexidade de tempo $O(n^2)$: para todos os pares de segmentos encontre, se existir, um ponto de intersecção entre eles. Entretanto, um tal algoritmo não seria muito útil na prática, onde o número de intersecção, em geral, não é grande. Portanto, parece razoável procurarmos um algoritmo que seja *output-sensitive*, ou seja, um algoritmo cuja complexidade de tempo seja, além de função do tamanha da entrada, também leve em consideração o tamanha da saída.

Denotaremos por I o número de intersecções entre o conjunto de segmentos dados. Em parte da nossa exposição suporemos, por simplicidade, que o conjunto de segmentos de reta dados está em *posição geral*, ou seja, não estaremos preocupados com os casos onde três ou mais segmentos se intersectam em um mesmo ponto.

Date: 1 de abril de 2001.

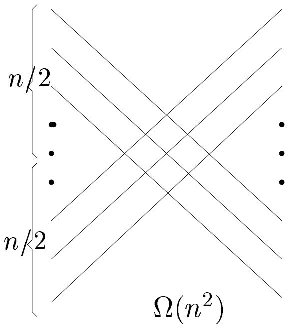


Figura 1. Exemplo onde existem $\Omega(n^2)$ intersecções entre n segmentos.

Na Seção 2, discutiremos sobre como determinar o ponto de intersecção, se existir, entre dois segmentos não paralelos — em geometria computacional é típico não discutirmos sobre este tópico, que é em essência mais próximo, talvez, de análise numérica. Na Seção 3, veremos como podemos decidir se dois segmentos se intersectam ou não sem termos que calcular o ponto de intersecção. Uma versão unidimensional mais simples do problema da intersecção de segmentos será tratada na Seção 4, ou seja, trataremos do problema de determinar se existem *intervalos* que se sobrepõem entre um conjunto de intervalos dados — essa versão mais simplificada, a exemplo do que ocorreu quando estudamos o Problema do Par Mais-Próximo, nos dará idéias para obtermos um algoritmo eficiente para a versão bidimensional do problema. Na Seção 5, apresentaremos o Método do Plano de Varreduro, que será usado na Seção 6 para obtermos um algoritmo de complexidade de tempo $O(n \log n + I \log n)$ que determina todas as intersecções de um conjunto de segmentos dados. A Seção 7 discute a cota inferior para o Problema da intersecção de segmentos. Finalmente, na Seção 8, temos uma série de exercícios.

Observação. Os tópicos que cobriremos a seguir foram extraídos do Capítulo 2 de De Berg, Van Kreveld, Overmars e Schwarzkoft [7], Capítulo 35 de Cormen, Leiserson e Rivest [6], Capítulo 7 de Laszlo [9], Capítulos 1 e 7 de O'Rourke[12], das notas de aula de Mount [10], Capítulo 7 de Preparata e Shamos [14], e Capítulo 5 de Resende e Stolfi [8].

2. Cálculo do ponto de intersecção entre dois segmentos

Na próxima seção veremos como podemos determinar se dois segmentos dados se intersectam sem de fato calcular o ponto de intersecção. Trataremos nesta seção do problema: *dados:* dois segmentos de reta ab e cd; *encontrar:* um ponto de intersecção entre ab e cd, se existir. Suponha que

$$a = (a_x, a_y), \quad b = (b_x, b_y), \quad c = (c_x, c_y), \quad \text{e} \quad d = (d_x, d_y).$$

Um segmento de reta será dado através das coordenadas de seus pontos extremos.

2.1. **Equações de inclinação (*slope equations*).** Uma possível idéia para determinarmos a intersecção entre os segmentos ab e cd é procedermos como segue:

- calculamos a equação da reta l_{ab} que contém ab;
- calculamos a equação da reta l_{cd} que contém cd;
- determinarmos um $p := l_{ab} \cap l_{cd}$;
- testamos se p pertence a ab e a cd.

Seja $y = m_{ab}x + b_{ab}$ a equação da reta l_{ab}. Podemos calcular o coeficiente angular m_{ab} de l_{ab} da seguinte maneira:

$$m_{ab} = \frac{b_y - a_y}{b_x - a_x} \quad \text{e} \quad b = m_{ab}a_x - a_y.$$

Aqui vemos um primeiro problema, se $a_x = b_x$, ou seja, se o segmento ab for vertical temos uma divisão por zero no cáculo acima. Isto significa que a implementação desta idéia terá que tratar separadamento do caso 'segmento vertical' (caso especial para a implementação). Note que este caso especial não é geométrico, no sentido que a 'culpa' é do nosso sistema de coordenadas ; se rotacionacimos um pouco o nosso sistema de coordenadas não teriamos este problema.

Suponha que obtivemos as equações $y = m_{ab}x+b_{ab}$ de l_{ab} e $y = m_{cd}x+b_{cd}$ de l_{cd}. O ponto de intersecção (x, y) das retas l_{ab} e l_{cd} satisfaz

$$y = \frac{b_{ab}m_{cd} - b_{cd}m_{ab}}{m_{cd} - m_{ab}} \quad \text{e} \quad x = \frac{b_{ab} - b_{cd}}{m_{cd} - m_{ab}}.$$

Se $m_{ab} = m_{cd}$, ou seja, se as retas l_{ab} e l_{cd} forem paralelas, teremos uma divisão por zero. Este é mais um caso especial que deve ser tratado por uma implementação desta idéia. Este é um caso especial genuinamente geométrico, restas paralelas existem na natureza.

2.2. **Representação paramétrica.** Determinar a coordenada do ponto de intersecção envolve a solução de um pequeno sistema de equações lineares. Uma das maneiras mais naturais de fazermos isto é considerarmos um *representação paramétrica* dos segmentos. Recordemos que qualquer ponto do segmento ab pode ser escrito como combinação convexa dos pontos a e b, combinação essa que envolve um parametro real λ:

$$p(\lambda) = (1 - \lambda)a + \lambda b, \quad \text{para } 0 \leq \lambda \leq 1.$$

De maneira similar, podemos utilizar um parametro μ para representar parametricamente um ponto arbitrário do segmento cd:

$$q(\mu) = (1 - \mu)c + \mu d, \quad \text{para } 0 \leq \mu \leq 1.$$

Uma intersecção entre os segmentos ab e cd ocorre se e somente se podemos encontrar λ e μ no intervalo $[0, 1]$, tais que $p(\lambda) = q(\mu)$. Portanto, para encontrarmos tais parâmetros, basta resolvermos o seguinte sistema de duas equações:

$$\begin{aligned}(1 - \lambda)a_x + \lambda b_x &= (1 - \mu)c_x + \mu d_x \\ (1 - \lambda)a_y + \lambda b_y &= (1 - \mu)c_y + \mu d_y\end{aligned}$$

Encontrar λ e μ que satisfaçam o sistema acima é um exercício simples de álgebra linear (lembre-se da Regra de Cramer). Temos que

$$\begin{aligned}\lambda &= [a_x(d_y - c_y) + c_x(a_y - d_y) + d_x(c_y - a_y)]/D,\\ \mu &= [a_x(c_y - b_y) + b_x(a_y - c_y) + c_x(b_y - a_y)]/D,\\ D &= a_x(d_y - c_y) + b_x(c_y - d_y) + d_x(b_y - a_y) + c_x(a_y - b_y).\end{aligned}$$

Divisão por zero pode ocorrer nos cáculos acima. A situação em que $D = 0$ corresponde ao caso em que os segmentos ab e cd são paralelos (possivelmente colineares). Estes casos especiais devem ser tratados com cuidado. Se $D \neq 0$ então obteremos valores para λ e μ como números racionais (se as coordenadas dos pontos forem inteiros então λ e μ são razões de números inteiros). Podemos aproximar estes números por números em ponto flutuante (*floating point*) ou se desejarmos valores exatos podemos simular números racionais exatamente usando 'inteiros grandes' (high-precision integers) multiplicando-os pelo mínimo multiplo comum dos números envolvidos. Uma vez que os valores de λ e μ foram obtidos, precisamos apenas verificar se eles são números no intervá-lo $[0, 1]$ — se for o caso então encontramos o ponto de intersecção entre os segmentos, caso contrário os segmentos não se intersectam.

Uma implementação da idéia acima está logo a seguir. Note que o código não determina se dois segmentos paralelos se intersectam. O código foi escrito por J. O'Rourke [12].

```
#define X 0
#define Y 1
typedef enum {FALSE, TRUE} bool;
#define DIM 2
typedef int tPointi [DIM]
typedef double tPointd [DIM]

bool SegSegInt (tPointi a, tPointi b, tPointi c, tPointi d, tPointi p) {
    double s, t;            /* parâmetros da equações */
    double num, denom;          /* numerador e denominador */

    denom = a[X] * (d[Y] - c[Y]) + b[X] * (c[Y] - d[Y]) +
            d[X] * (b[Y] - a[Y]) + c[X] * (a[Y] - b[Y]) ;

    /* Se denom = 0 os segmentos são paralelos */
    if (denom == 0) return FALSE ;

    num = a[X] * (d[Y] - c[Y]) + b[X] * (a[Y] - c[Y]) +
          d[X] * (c[Y] - a[Y]);
    s = num / denom;

    num = - (a[X] * (c[Y] - b[Y]) + b[X] * (a[Y] - c[Y]) +
            c[X] * (b[Y] - a[Y]));
    t = num / denom;
```

```
    p[X] = a[X] + s * (b[X] - a[X]);
    p[Y] = a[Y] + s * (b[Y] - a[Y]);

    if ((0.0 ≤ s) ∧ (s ≤ 1.0) ∧ (0.0 ≤ t) ∧ (t ≤ 1.0))
        return TRUE;
    else
        return FALSE;
}
```

2.3. **Filosofia em relação a casos especiais.** Podemos tomar as seguintes atitudes com relação a casos especiais:

(1) (Ignorá-los). Supomos que a entrada é restrita a instâncias em que casos especias não ocorrem. Faremos isto algumas vezes durante o andamento da disciplina para não nos desviarmos a atenção dos pontos mais importantes.
(2) Escrever programas que tratam dos casos especiais. Algumas vezes não temos outra saída além desta, mas devemos ter em mente que tratar casos especiais e algo difícil e tende a propagar erros. Mas tudo bem, ninguém disse que fazer implementações é algo fácil (se disse foi um engano).
(3) Encontrar um sistema geométrico no qual os casos especiais desapareçam. Se posível, esta opção é ótima, por exemplo, no cálculo da inteserção de segmentos em coordenadas homogêneas não temos o perigo de *divide by zero.*
(4) Encontrar uma representação algorítmica das mesmas idéias geométricas e que evitam casos especiais. (Não sem bem o que é isto ...)

3. Detecção da intersecção entre dois segmentos

Veremos nesta seção como podemos decidir se dois segmentos de reta dados ab e cd possuem uma intersecção sem de fato calcular um ponto de intersecção. Novamente consideraremos que

$$a = (a_x, a_y), \quad b = (b_x, b_y), \quad c = (c_x, c_y), \quad \text{e} \quad d = (d_x, d_y).$$

e que um segmento de reta será dado através das coordenadas de seus pontos extremos.

O algoritmo que veremos tem duas fases. Na Fase 1, que chamaremos de teste de rejeição rápida, fazeremos testes, que são simples comparações, envolvendo as coordenadas dos extremos dos segmentos. Caso os segmentos não passem por este teste eles não tem a menor chance de se intersectarem. Caso os segmentos passem iremos então para a Fase 2 onde, usando o fato que os segmentos passaram pelo teste de rejeição rápida, decidiremos se eles se intersectam ou não. Nesta segunda fase faremos uso de dois predicados geométricos, a saber:

Left(u,v,w): Este predicado recebe como parâmetro 3 pontos e é verdadeiro se e somente se o ponto w está à esquerda do segmento orientado $\vec{uv}$. Note que este predicado pode ser computado em tempo constante simplesmente calculando-se o sinal de $\mathcal{A}(T)$, onde T é o triângulo $\triangle(u, v, w)$ e $\mathcal{A}(T)$ denota a área sinalizada do triângulo T. Resumindo, `Left`(u, v, w) é verdadeiro se e somente se $\mathcal{A}(T) > 0$.

Collinear(u,v,w): Este predicado recebe como parâmetro 3 pontos e é verdadeiro se e somente se os ponto u, v e w são colineares. Para computarmos este predicado basta determinarmos se $\mathcal{A}(T) = 0$, onde T é o triângulo $\triangle(u, v, w)$.

Para ver uma implentação em `C` destes predicados veja a 'capítulo' Primitivas que está na página das notas de aula.

Notemos que a saída do algoritmo que veremos será um simples SIM ou NÃO, significando que os segmentos se intersectam ou não. Ou seja, o problema que estamos considerando aqui é um problema de decisão.

3.1. **Fase 1: rejeição rápida.** Dois segmentos não podem se intersectar se os retângulos determinados pelos seus estremos não se intersectam, veja Figura 2.

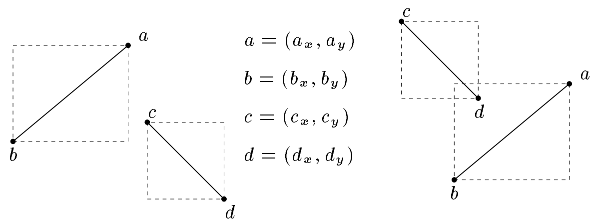


FIGURA 2. Retângulos determinados por dois segmentos.

Sejam

$$a'_x = \min\{a_x, b_x\},\ a'_y = \min\{a_y, b_y\},\ b'_x = \max\{a_x, b_x\},\ b'_y = \max\{a_y, b_y\}$$
$$c'_x = \min\{c_x, d_x\},\ c'_y = \min\{c_y, d_y\},\ d'_x = \max\{c_x, d_x\},\ d'_y = \max\{c_y, d_y\}.$$

Veja Figura 3.

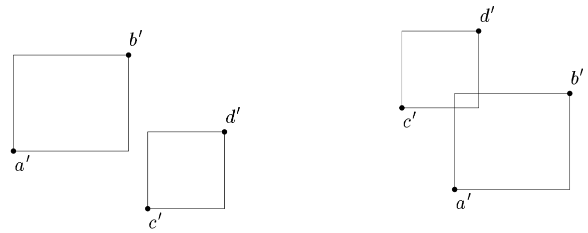


FIGURA 3. Extremos dos retângulos.

Os retângulos determinados pelos segmentos ab e cd se intersectam se e somente se a seguinte condição é satisfeira (verifique!):

$$(a'_x \leq d'_x) \wedge (c'_x \leq b'_x) \wedge (a'_y \leq d'_y) \wedge (c'_y \leq b'_y).$$

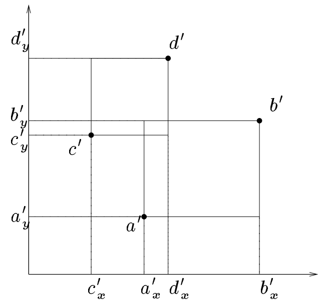


FIGURA 4. Ilustração do teste da Fase 1.

3.2. **Fase 2.** Esta fase é baseada no seguinte fato. Dois segmentos de reta dados ab e cd se intersectam se e somente se:

- ab e cd passaram pelo teste de rejeição; **e**
- os pontos a e b estão em lados opostos do segmento cd **e** os pontos c e d estão em lados opostos do segmento ab **ou** os pontos a, b, c e d são colineares.

As possíveis situações estão ilustradas na Figura 5.

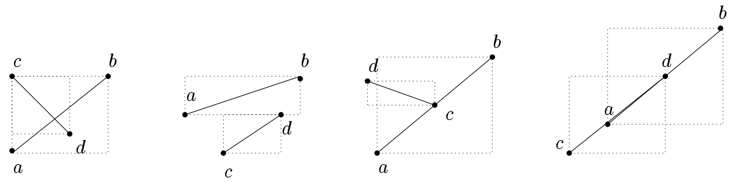


FIGURA 5. Possíveis situações.

Assim, esta fase se resume ao seguinte teste:

$$[(\texttt{Left}(a,b,c) \neq \texttt{Left}(a,b,d)) \wedge (\texttt{Left}(c,d,a) \neq \texttt{Left}(c,d,b))]$$
$$\vee$$
$$[\texttt{Collinear}(a,b,c) \wedge \texttt{Collinear}(a,b,d)].$$

Notemos que como cada predicado pode ser computado em tempo constantes então a complexidade de tempo do algoritmo para teste de intersecção também é constante.

EXERCÍCIO 1. Dê uma prova cuidadosa de que o algoritmo proposto para testar a intersecção entre dois segmentos de reta dados funciona.

EXERCÍCIO 2. Implemente em `C` o algoritmo de teste de intersecção que acabamos de obter.

4. ALGORITMO PARA INTERSECÇÃO DE INTERVALOS

Dados n intervalos na reta podemos decidir em tempo $O(n^2)$ se algum par se sobrepõe, ou seja, se existe alguma intersecção entre algum par de segmentos.

Uma outra idéia para determinarmos se dois segmentos se sobrepõem é a seguinte. Denotemos por e_i e d_i os extremos esquerdo e direito do segmento s_i $(i = 1, \dots, n)$, respectivamente. Podemos ordenar os intervalos $s_1, \dots, s_n$ em relação aos seus extremos mais à esquerda. Assim, depois de gastarmos tempo $\Theta(n \log n)$ com a ordenação dos segmentos podemos supor que $e_1 \leq e_2 \leq \dots \leq e_n$ (veja Figura 6). Se os $e's$ das extremidades

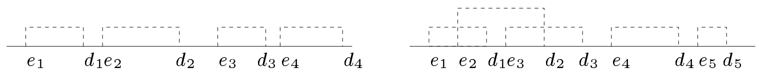


Figura 6. Intervalos.

esquerdas e os $d's$ das extremidades direitas se alternam então o segmentos não se intersectam, caso contrário existe uma sobreposição entre um par de segmentos. Assim, após ordenarmos o segmentos, podemos percorrer a reta (geometricamente falando) do extremo mais à esquerda até o extremo mais à direito e durante nosso percurso precisamos apenas lembrar se o último extremo encontrado foi um dos $e's$ ou foi um dos $d's$.

Algoritmo `Varredura1`(S)
`Entrada:` uma coleção $S = \{s_1, \dots, s_n\}$ de n intervalos de $\mathbb{R}$.
`Saída:` SIM se um par de intervalos se sobrepõem e NÃO caso contrário.
1. Sejam $p_1, \dots, p_{2n}$ os pontos extremos de $s_1, \dots, s_n$ ordenados de tal forma que $p_1 \leq p_2 \leq \dots \leq p_{2n}$
2. intervalos $\leftarrow 0$;
   /* Um ponto x percorre (varre) a reta da esquerda para adireita */
3. **for** $x = p_1, \dots, p_{2n}$ **do**
4.   **if** $x \in \{d_1, \dots, d_n\}$ **then**
5.   intervalos $\leftarrow$ intervalos $-$ 1;
6.   **else if** $x \in \{e_1, \dots, e_n\}$ **then**
7.   intervalos $\leftarrow$ intervalos $+1$;
8.   **if** intervalos ≥ 2 **then return** "Os intervalos se sobrepõem";
9. **return** "Os intervalos não se sobrepõem".

A complexidade de tempo deste algoritmo é $\Theta(n \log n)$.

Podemos entender o algoritmo acima como um ponto x que vai varrendo a (andando pela) reta da esquerda para a direita e este ponto mantém informações sobre se, ou qual, o(s) segmentos ele pertence. Isto sugere que para estendermos as idéias apresentadas nesta seção para o problema bidimensional precisamos de uma relação de ordem entre os segmentos e de estruturas de dados mais sofisticadas para guardar o 'estado' do ponto. Nas próximas seções veremos como este tipo de idéia pode ser generalizada para obtermos um algoritmo *output-sensitive* para o problema da intersecção de segmentos no plano.

5. Método da linha de varredura

Em algoritmos por linha-de-varredura (*plane-sweep algorithms*) uma linha (ou reta) imaginária, digamos vertical, move-se da esquerda para a direita.

A medida que a linha prossegue o problema restrito aos objetos que ficaram à esquerda da linha é resolvido. Uma *estrutura da linha de varredura* (*sweepline structure* ou *sweepline status*) mantem o 'estado' do algoritmo para cada posição da linha de varredura. Esta estrutura guarda toda a informação sobre os dados do problema que estáão à esquerda da linha de varredura e que serão necessários para estender para o lado direito da linha a solução parcial corrente do problema.

A linha de varredura avança em incrementos, parando em posições chaves chamadas de *pontos eventos* (*event points*). Esses pontos, quando alcançados pela linha, fazem com que o estado do algoritmo mude. Pontos eventos são mantidos em uma *fila de eventos* (*event queue* ou *event-point schedule*), que é uma lista ordenada dos pontos eventos de acordo a ordem que estes devem ser processados: as x-coordenadas dos pontos é usada para obtermos o próximo ponto evento que será alcançado pela linha e que deverá ser tratado pelo algoritmo. Em algumas aplicações a fila de eventos é *estática*, ou seja, ela está completamente definida no instante inicial do algoritmo, em outras aplicações ela é *dinânica*, isto é, novos pontos eventos podem ser inseridos na fila durante a execução do algoritmo (este será o caso do algoritmo que veremos mais no final deste capítulo).

Um algoritmo por linha-de-varredura passa por transições em cada ponto evento. As transições consistem tipicamente de três tipos de ações:

(1) (atualizar a fila de eventos) Remover o ponto evento corrente, junto com qualquer outro que se tornou obsoleto e, eventualmente, inserir novos pontos eventos na fila de eventos.
(2) (atualizar a estrutura de linha de varredura) Atualizar estruturas de dados que mantém a estrutura da linha de varredura para que esta represente a situação atual, que é devida a nova posição da linha.
(3) (resolver problema) Resolver mais do problema, estendento a solução corrente.

O método da linha de varredura reduz um problema estático bidimensional para um problema dinâmico unidimensional (o problema de manter a estrutura da linha). O problema unidimensional resultante é geralmente mais simples que o problema bidimensional original (pelo menos é o que se deseja...). Nesta disciplina veremos várias aplicações deste método, em particular, para determinarmos todas as intersecção entre um conjunto de segmentos e para particionarmos um polígono em polígonos monótonos.

6. Algoritmo do tipo linha-de-varredura para intersecção de segmentos

O algoritmo que estudaremos nesta seção foi apresentado por Bentley e Ottmann [2] em 1979. Alguns anos antes Shamos e Hoey [15] propuseram um algoritmo de complexidade de tempo $O(n \log n)$ que decide se existe intersecção entre algum dos pares de segmentos dados, onde n é o número de segmentos.

Seja $S := \{s_1, \dots, s_n\}$ o conjunto de segmentos para os quais queremos computar todas as intersecções. Uma idéia natural para desenvolvermos um algoritmo eficiente é evitarmos testar a intersecção entre pares de segmentos que estão 'longes' um do outro. Mas como podemos fazer isto? Vamos primeiro eliminar os casos fáceis. Defina o *x-intervalo* de um segmento como sendo a sua projeção no eixo das abscissas. Quando os x-intervalos de dois segmentos não se sobrepõem então eles não tem a menor chance de se intersectarem e, portanto, não precisamos testar intersecção entre eles (veja Figura 7). Assim, só precisamos realizar o teste de intersecção entre pares de segmentos cujos x-intervalos se sobrepõem, isto é, segmentos para os quais existe um *linha vertical* que intersecta ambos os segmentos. Para determinarmos estes pares de segmentos usaremos o método da linha de varredura, isto é, imaginaremos uma linha vertical varrendo (percorrendo) o plano da esquerda para a direita. Enquanto a linha varre o plano nós manteremos todos os segmentos intersectados por ela; os candidatos a realizarmos o teste de intersecção.

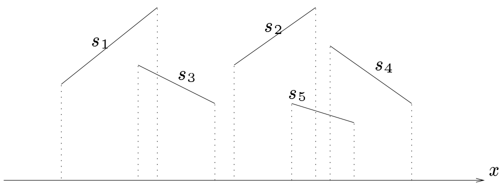


FIGURA 7. x-intervalos.

Denotaremos a linha de varredura por l. A estrutura ou status (*structure* ou *status*) da linha é o conjunto de segmentos que ela intersecta. A estrutura de l só muda em alguns pontos específicos, isto é, a estrutura só se altera em momentos *discretos*; apesar de, pelo menos na nossa imaginação, a linha estar se movendo continuamente. Os pontos nos quais a estrutura de l muda são os *pontos eventos* que (pelo menos até agora) consistem dos extremos dos segmentos de l.

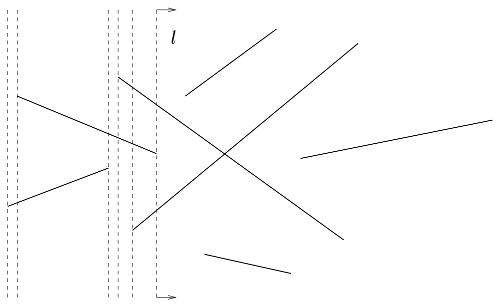


FIGURA 8. Linha de varredura em movimento. Os pontos onde a estrutura da linha é alterada são os extremos dos segmentos.

Infelizmente, fazer o teste de intersecção entre cada par de segmentos que são simultaneamente intersectados pela linha de varredura l não é suficiente para obtermos um algoritmo rápido, ou seja, um algoritmo de complexidade de tempo $o(n^2)$ para conjunto de segmentos com poucas intersecção. A linha l pode estar intersectando $\Omega(n)$ segmentos simultaneamente e nós não queremos fazer $\Omega(n^2)$ testes de intersecções envolvendo todos os segmentos intersectados pela linha, a não ser que seja necessário. A idéia aqui é notarmos que basta realizarmos o teste intersecção entre pares de segmentos que ficam adjacentes em l em algum momento. Logo adiante tornaremos esta idéia mais precisa.

Existe um número grande de casos especiais que podem obscurecer os ingredientes principais do algoritmo que veremos. Assim, faremos uma série de hipóteses sobre a entrada, que podem ser removidas a custa de algum cuidado extra no tratamento desses casos. Suporemos por enquanto que

(1) nenhum segmento é vertical;
(2) se dois segmentos se intersectam então eles se intersectam em um único ponto (ou seja, os segmentos são não-colineares); e
(3) três segmentos não se intersectam em um mesmo ponto.

Diremos que dois segmentos s e s' são comparáveis na abscissa u se a reta vertical l_u passando por $(u, 0)$ intersecta ambos, s e s'. Suponha que $p = (p_x, p_y) := s \cap l$ e $p' = (p'_x, p'_y) := s' \cap l$, então escreveremos $s >_u s'$ se $p_y > p'_y$ (veja Figura 9).

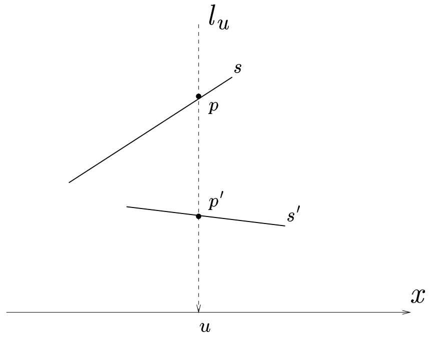


FIGURA 9. Exemplo onde $s >_u s'$.

Na Figura 10 vemos que $s_1 >_r s_3$, $s_1 >_t s_2$, $s_2 >_t s_3$, $s_1 >_t s_3$, $s_2 >_u s_3$ e s_4 é incomparável com os demais segmentos —o x-intervalo de s_4 é disjunto dos x-intervalos dos demais segmentos.

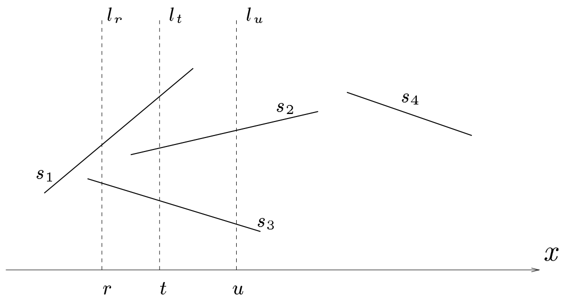


FIGURA 10. Ilustração das relações de ordem.

Para evitar testes de intersecção desnecessários a nossa estratégia será testar a intersecção somente entre pares de segmentos que fiquem consecutivos na linha de varredura l em algum momento. Uma questão natural é se isto é suficiente. Em particular, se dois segmentos se intersectam, existe algum momento em que esses segmentos são consecutivos na ordem induzida pela linha de varredura? Felizmente este é precisamente o caso.

Lema 1. *Sejam s e s' dois segmentos que se intersectam em um único ponto $p = (p_x, p_y)$ no interior relativo de ambos os segmentos. Suponha que nenhum outro segmento contém o ponto p. Então existe uma posição da linha de varredura l à esquerda de p (i.e. antes da linha alcançar p) tal que s e s' são adjacentes na linha (e portanto serão testados pelo algoritmo).*

Prova. Da nossa hipótese de posição geral segue que não existem três segmentos que se intersectam em um mesmo ponto. Portanto, existe um posição da linha de varredura, imediatamente à esquerda do ponto de intersecção, em que os segmentos que se intersectam são consecutivos em relação à ordem induzida pela linha de varredura. Considere o ponto evento q com a maior x-coordenada que é estritamente menor que p_x. A ordem dos segmentos induzida pela linha de varredura, logo após processarmos o ponto evento q, é identica a ordem dos segmentos induzida pela linha de varredura l imediatamente antes desta alcançar o ponto p e, portanto, s e s' são adjacentes neste ponto (veja Figura 11). ∎

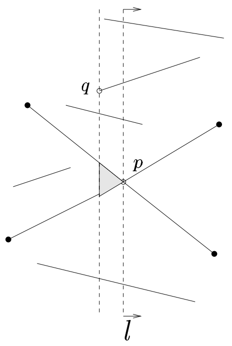


FIGURA 11. Ilustração da prova do Lema 1.

Assim, a nossa idéia está correta, pelo menos quando ignoramos os casos especiais mencionados anteriormente.

O que acontece quando a linha de varredura passa por um ponto de intersecção p entre dois segmentos s e s'? Pela Figura 11 podemos perceber que a ordem relativa entre s e s' se inverte.

6.1. **O algoritmo.** Podemos agora prosseguir e descrever o nosso algoritmo por linha-de-varredura. Os principais ingredientes do algoritmo estão logo abaixo.

Sweepline structure: A estrutura da linha de varredura l deve manter os segmentos que ela intersecta ordenados de acordo com a ordem induzida por l. Um segmento deverá ser inserido na estrutura da linha de varredura quando a sua extremidade esquerda for alcançada pela linha e removido quando a sua extremidade direita for alcançada. Sempre que segmentos ficam consecutivos na ordem da linha de varredura devemos fazer o teste de intersecção entre eles.

Event queue: A fila de pontos eventos do algoritmo será dinâmica. Inicialmente esta fila conterá todos os pontos extremos dos segmentos ordenados de acordo com as suas x-coordenadas, em caso de empate o ponto com menor y-coordenada deverá vir primeiro na fila (estamos ordenando os eventos em ordem léxica). Cada ponto evento contém informações auxiliares tais como o tipo do ponto evento (ponto extremo ou de intersecção) e quais os segmentos envolvidos. A medida que os pontos vão sendo processados os pontos de intersecção obtidos entre pares de segmentos deverão ser inseridos na fila de pontos eventos. Teremos assim três tipos de pontos eventos, a saber:

- ponto extremidade esquerda de segmento (*segment left endpoint*);
- ponto extremidade direita de segmento (*segment right endpoint*);
- ponto intersecção de segmentos (*intersection point*).

Event updates: Quando um evento é encontrado devemos atualizar a estrutura de dados associada a ele. A maneira que esta atualização é feita depende do tipo do ponto evento.

extremidade esquerda de segmento: O correspondente segmento s deve ser inserido na estrutura da linha de varredura l, baseado na ordem induzida por l (ou seja, de acordo com a y-coordenada do ponto evento que está sendo tratado). Devemos testar intersecção entre o segmento inserido e os segmentos sucessor e predecessor de s na ordem induzida por l; (Dois testes de intersecção são feitos).

extremidade direita de segmento: O correspondente segmento s deve ser removido da estrutura da linha de varredura l e o seu sucessor e predecessor na ordem induzida por l devem ser testados para determinarmos se eles se intersectam. (Um teste de intersecção é feito.)

intersecção de segmentos: A ordem relativa entre os segmentos que determinam o ponto de intersecção deve ser invertida na estrutura da linha de varredura l. Devemos fazer o teste de intersecção entre o novo segmento 'mais alto' e o seu sucessor na ordem induzida por l. Devemos fazer o teste de intersecção entre o novo segmento 'mais baixo' e o seu predecessor na ordem induzida por l. (Dois testes de interrsecção são feitos)

Qualquer ponto de intersecção determinado durante o processamento de um dos eventos acima deve ser inserido na fila de eventos, a não ser que ele já esteja na fila.

Ao longo da execução do algoritmo teremos três tipos de segmentos: segmentos *mortos* que são segmentos que estão totalmente à esquerda da linha de varredura, segmentos *ativos* que têm intersecção com a linha de varredura; e segmentos *adormecidos* que estão totalmente à direita da linha de varredura.

A estrutura da linha de varredura deve ser representada através de uma estrutura de dados $\mathcal{T}$ que admite as seguintes operações:

Insert$(s, \mathcal{T})$**:** insere o segmento s na ordem total mantida por $\mathcal{T}$;
Delete$(s, \mathcal{T})$**:** remove s de $\mathcal{T}$;
Above$(s, \mathcal{T})$**:** devolve o nome do segmento imediatamente acima de s na ordem mantida por $\mathcal{T}$;
Below$(s, \mathcal{T})$**:** devolve o nome do segmento imediatamente abaixo de s na ordem mantida por $\mathcal{T}$

Cada uma destas operações pode ser executada em tempo $O(\log n)$ se usarmos uma árvore balanceada de busca (AVL, árvore 2-3, rubro-negra) para representar $\mathcal{T}$, onde n é o numero de segmentos dados. Na Figura 12 vemos uma ilustração da execução do algoritmo.

A fila de eventos $\mathcal{Q}$ deve ser representada por uma estrutura de dados que mantenha os pontos eventos ordenados de acordo com as suas x-coordenadas, em caso de empate o ponto com menor y-coordenada deve aparecer na frente, e admita as seguintes operação:

Min$(\mathcal{Q})$**:** devolve o ponto evento que está na primeira posição da fila $\mathcal{Q}$ e remove-o de $\mathcal{Q}$;
Insert$(p, \mathcal{Q})$**:** insere em $\mathcal{Q}$ o ponto evento p;
Member$(p, \mathcal{Q})$**:** determina se o ponto p é um membro de $\mathcal{Q}$.

Uma estrutura de dados fila de prioridades, como d-heap, *leftist-heap*, *Fibonacci heap*, é capaz de realizar as operações `Min` e `Insert` em tempo $O(\log m)$ onde m é o número máximo de elementos na fila ao longo da execução do algoritmo. Não usaremos uma destas estruturas para representar $\mathcal{Q}$ pois elas não são capazes de executar a operação `Member` em tempo logarítmo. Manteremos $\mathcal{Q}$ em uma árvore balanceada de busca que pode executar todas as operações acima em tempo logarítmico.

Estamos agora preparados para apresentar uma descrição completa do algoritmo. A medida que a linha de varredura avança de ponto evento para ponto evento a estrutura da linha de varredura e a fila de pontos eventos são atualizadas e segmentos que se tornam adjacentes na estrutura $\mathcal{T}$ são submetidos ao teste de intersecção. Se a intersecção é determinada pela primeira vez o algoritmo devolve esta intersecção e ela é inserida na fila de eventos. Na Figura 12 pode ser vista uma ilustração da execução do algoritmo.

Algoritmo `Varredura2`(S)
`Entrada:` uma coleção $S = \{s_1, \ldots, s_n\}$ de n segmentos em $\mathbb{R}^2$.
`Saída:` todos os pontos de intersecção entre pares de segmentos
e junto cada ponto os segmentos que o contém.
1. $\mathcal{Q} \leftarrow \emptyset$; /* inicializa a fila de eventos */
2. Insira os pontos extremos dos segmentos em $\mathcal{Q}$ (`Insert`$(p, \mathcal{Q})$).
3. $\mathcal{T} \leftarrow \emptyset$;
4. **while** $\mathcal{Q}$ é não-vazia **do**
5. Determine o próximo ponto evento p em $\mathcal{Q}$ e remova-o de $\mathcal{Q}$ (i.e. `Min`$(\mathcal{Q})$).
6. `TrataPontoEvento`(p);

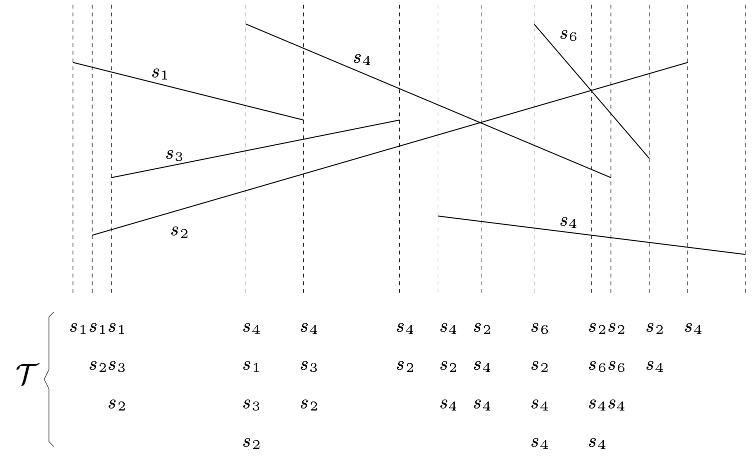


FIGURA 12. Simulação da execução do algoritmo exibindo a ordem dada pela estrutura T da linha de varredura.

Já discutimos como os pontos eventos devem ser tratados. O próximo algoritmo descreve, de uma manieira um pouco mais completa, como eventos serão tratados.

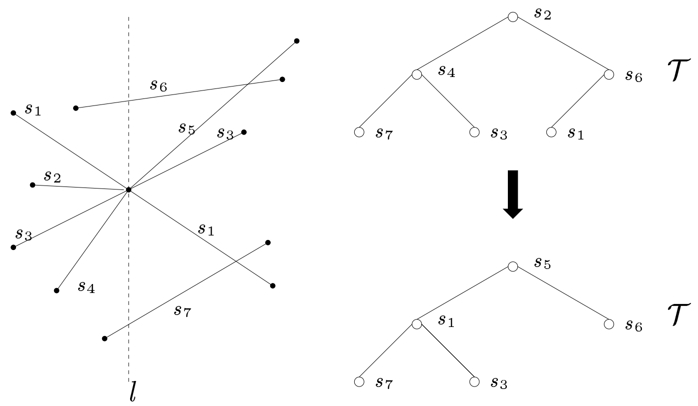


FIGURA 13. Um ponto evento e a mudança causada na estrutura da linha de varredura $\mathcal{T}$ após o processamento deste ponto.

Algoritmo `TrataPontoEvento`(p)

1. Seja $E(p)$ o subconjunto dos segmentos cuja extremidade esquerda é p; estes segmentos são armazenados junto com p. Para segmentos verticais a sua extremidade esquerda é a de menor y-coordenada.
2. Encontre todos os segmentos armazenados em $\mathcal{T}$ que contém p; eles são adjacentes em $\mathcal{T}$. Seja $D(p)$ o subconjunto de segmentos encontrados cuja extremidade direita é p, e seja $C(p)$ o subconjunto dos segmentos encontrados que contém p em seu interior.
3. **if** $E(p) \cup D(p) \cup C(p)$ contém mais do que um segmento
4.   **then** devolva p junto com $E(p) \cup D(p) \cup C(p)$.

5. Remova os segmentos em $D(p) \cup C(p)$ de $\mathcal{T}$.
6. Insira os segmentos em $E(p) \cup C(p)$ e, $\mathcal{T}$. A ordem dos segmentos em $\mathcal{T}$ deve corresponder a ordem em que eles são intersectados por l imediatamente após p. Se existe um segmento vertical, ele deve vir depois de todos os segmentos contendo p.
7. /* Remoção e reinsersão de segmentos em $C(p)$ inverte a ordem relativa. */
8. **if** $E(p) \cup C(p) = \emptyset$
9.   **then** Seja s_e e s_d os segmentos à esquerda e direita de p em $\mathcal{T}$.
10.   `EncontreNovosEventos`(s_e, s_d, p)
11.   **else** Seja s' o segmento mais à esquerda de $E(p) \cup C(p)$ em $\mathcal{T}$.
12.   Seja s_e o segmento à esquerda (predecessror) de s' em $\mathcal{T}$.
13.   `EncontreNovosEventos`(s_e, s', p)
14.   Seja s'' o segmento mais à direita de $E(p) \cup C(p)$ em $\mathcal{T}$.
15.   Seja s_r o segmento à direita (sucessor) de s'' em $\mathcal{T}$.
16.   `EncontreNovosEventos`(s'', s_d, p)

Note que na linhas 8–16 estamos supondo que s_e e s_r existem. Se eles nao existem o passos correspondenes não devem ser executados.

A função `EncontreNovosEventos` está logo a seguir.

Algoritmo `EncontreNovosEventos`(s_e, s_d, p)

1. **if** s_e e s_d se intersectam à direita da linha de varredura, ou sobre ela e a direita do ponto evento corrente p, e a intersecção ainda não está presente em $\mathcal{Q}$
2.   **then** insira o ponto de intersecção como um evento em $\mathcal{Q}$.

6.2. **Corretude do algoritmo.** É claro que o algoritmo devolve pontos de intersecção, mas não esta claro que o algoritmo devolve *todos* eles.

Teorema 2. *o Algoritmo `Varredura2` devolve todos os pontos de intersecção e, para cada ponto, a lista dos segmentos que o contém.*

Prova. A fila de eventos mantém os pontos em ordem léxica de acordo com as x-coordenadas e y-coordenadas. Provaremos o teorema por indução na prioridade dos pontos eventos.

Seja p um ponto de intersecção e suponha que todos os pontos de intersecção com prioridade maior já foram devolvidos pelo algoritmo corretamente. Provaremos que p e os segmentos que contém p serão computados corretamente. Seja $E(p)$ o subconjunto dos segmentos que que têm p como extremidade esquerda (ou, para segmentos verticais, o extremo mais baixo), seja $D(p)$ o subconjunto dos segmentos que têm p como extremidade direita (ou, para segmentos verticais, o extremo mais alto), e seja $C(p)$ o subconjunto dos segmentos que têm p em seu interior relativo.

Primeiro, suponha que p é extremo de um ou mais segmentos. Neste caso p é armazenado na fila de eventos $\mathcal{Q}$ no início do algoritmo. Os segmentos em $E(p)$ são armazenados em p, logo eles serão encontrados. Os segmentos em $D(p)$ e $C(p)$ estão armazenados em $\mathcal{T}$ quando o ponto p é processado, logo eles também serão encontrados na Linha 2 do algoritmo `TrataPonto Evento`. Portanto, p e todos os segmentos envolvendo p são determinados corretamente quando p é um ponto extremo de um ou mais segmentos.

Suponha agora que p não é um ponto extremo de algum segmento. Precisamos mostrar que p será inserido em $\mathcal{Q}$ em algum momento. Notemos que todos os segmentos que envolvidos com p tem p em seu interior relativo. Ordene estes segmentos ao redor de p de acordo com os seus ângulos, sejam s' e s'' dois segmentos vizinhos. Da prova do Lema 1 vemos que existe um ponto evento q à esquerda de p, com maior prioridade do que p, em que s' e s'' ficam adjacentes quando a linha de varredura passa por q. No Lema 1 supomos por simplicidade que s' e s'' não são verticais, mas não é difícil adaptar a prova para segmentos verticais. Por indução, o ponto evento q foi tratado corretamente, o que significa que p foi detectado em inserido em $\mathcal{Q}$. ∎

6.3. **Análise do algoritmo.** Agora que sabemos que o algoritmo está correto estudaremos a sua complexidade de tempo e espaço.

Teorema 3. *A complexidade de tempo do Algoritmo* `Varredura2` *para um conjunto S de n segmentos de reta no plano é $O(n \log n + I \log n)$, onde I é o número de pontos de intersecção se segmentos de S.*

Prova (esboço). O trabalho feito pelo algoritmo é dominado pelo tempo gasto em atualizar as várias estruturas de dados (caso contrário gastaríamos somente tempo constante com o processamento de cada evento). Precisamos estimar duas coisas: o número de operações aplicadas a cada estrutura de dados e a quantidade de tempo gasto para processar cada operação.

Existem no máximo n segmentos na estrutura $\mathcal{T}$ da linha de varredura em qualquer instante, e portanto o tempo necessário para executar qualquer operação em $\mathcal{T}$ é $O(\log n)$ — já que estamos representando $\mathcal{T}$ através de uma árvore de busca balanceada.

Como não permitimos eventos duplicados na fila de eventos, o número total de elementos na fila em qualquer instante é no máximo $2n + I$. Como usamos uma árvore de busca balanceada para representar a fila de eventos, cada operação gasta tempo máximo logarítmico no comprimento da fila, que é $O(\log(2n + I))$. Como $I \leq n^2$, a quantidade anterior é no máximo $O(\log n^2) = O(2 \log n) = O(\log n)$.

Cada evento está envolvido em um número constante de acessos ou operações para atualizar a estrutura da linha de varredura e a fila de eventos. Assim, devido ao parágrafo anterior temos que cada operação gasta tempo $O(\log n)$, donde segue que o tempo total para processar todos os eventos é

$$O((2n + I) \log n) = O((n + I) \log n) = O(n \log n + I \log n).$$

Logo, esta é a complexidade de tempo do algoritmo. ∎

Ainda temos que analisar a quantidade de espaço gasto pelo algoritmo. A estrutura $\mathcal{T}$ armazena um segmento no máximo uma vez, portanto o espaço gasto por $\mathcal{T}$ é $O(n)$. A fila $\mathcal{Q}$ pode ficar maior que $O(n)$. O algoritmo insere um ponto de intersecção em $\mathcal{Q}$ assim que a intersecção é detectada e o remove quando o ponto é processado. Quando o algoritmo fica um grande número de passos sem processar um ponto de intersecção pode ocorrer que que cresça bastante. ë claro que o espaço gasto por $\mathcal{Q}$ nunca será maior que $O(n + I)$, mas seria melhor se o espaço gasto fosse linear.

Existe um maneira relativamente simples de fazer com que os espaço gasto seja $O(n)$: somente armazena pontos de intersecção de pares de segmentos que são correntemente adjacentes na estrutura da linha de varredura. Procedendo desta maneira a fila $\mathcal{Q}$ nunca conterá mais que um número linear $(ouseja, O(n))$ de pontos eventos. Esta modificação requer que o algoritmo remova um ponto de intersecção da fila $\mathcal{Q}$ assim que o par de segmentos que se intersectam nesse ponto deixam de ser adjacentes. Estes segmentos ficarão adjacentes novamente antes da linha de varredura alcançar o ponto de intersecção entre eles, o que fará com que o ponto volte a ser inserido na fila. O tempo total gasto pelo algoritmo, após esta modificação, continua $O(n \log n + I \log n)$. Esta idéia para reduzir os espaço de trabalho do algoritmo é devida a Pach e Sharir [13]; Brown [3] descreveu um método alternativo para obter a redução.

Teorema 4. *Seja S um conjunto de n segmentos no plano. Todos os pontos de intersecção, e com eles os respectivos pares de segmentos, podem ser obtidos em tempo $O(n \log n + I \log n)$ e espaço $O(n)$, onde I é o número de pontos de intersecção.* ∎

7. Cota inferior

Veremos nesta seção que o problema de determinar todas as intersecções de um conjunto de n segmentos tem cota inferior $\Omega(n \log n + I)$, onde I é o número de intersecções entre os segmentos. Claramente qualquer algoritmo que determina todas as intersecções tempo complexidade de tempo de $\Omega(I)$, já que este é o tempo gasto para escrever a resposta. O que não é obvio é que tempo $\Omega(n \log n)$ é necessário.

Lembremos que o Problema da Unicidade de Elemento consiste em decidir se n números reais dados são todos distintos. Este problema tem, no modelo de árvore de decisão algébrica, cota inferior $\Omega(n \log n)$.

Dada um lista de n números reais $(x_1, \ldots, x_n)$ podemos, em tempo linear, contruir um conjunto de n segmentos vérticais, todos tendo Observemos que os números dados são distintos se e somente se os respectivos segmentos não intersectam. Portanto, se existisse um algoritmo que determina-se todas as intersecções em tempo $o(n \log n)$ (significa estritamente menor que $\Theta(n \log n)$) então poderiamos resolver o Problema da Unicidade de Elementos em tempo $o(n \log n)$, contrariando a cota inferior de $\Omega(n \log n)$ para o problema.

O algoritmo que vimos tem complexidade de tempo $O(n \log n + I \log n)$, portanto não é ótimo. Uma questão natural é se existe um algoritmo ótimo para o problema. De fato, existe um algoritmo de complexidade de tempo $O(n \log n + I)$ que determina todas as intersecções de um conjunto de n segmentos de reta que usa espaço $O(n + I)$ (cf. Chazelle e Edelsbrunner [4]). Mais tarde Clarkson e Shor [5] e Mulmuley [11] apresentaram um algoritmo probabilístico incremental que tem complexidade de tempo esperada de $O(n \log n + I)$, o espaço gasto por estes algoritmos é $O(n)$ e $O(n + I)$, repectivamente. Mais recentemente, Balaban [1] propôs um novo algoritmo determinístico para o problema de intersecção de segmentos que tem complexidade de tempo $O(n \log n + I)$ e usa espaço $O(n)$. Portanto, este é o primeiro algoritmo determinístico que é ótimo com relação a espaço e tempo.

Teorema 5. *O problema de determinar todas as intersecções de um conjunto de n segmentos de reta tem, no modelo de árvore de decisão algébrica, cota inferior $\Omega(n \log n + I)$, onde I é o número de intersecções entre os segmentos.* ∎

8. Exercícios

(1) (Exercício 2.1 de [7]) Seja S um conjunto de n segmentos de reta disjuntos cujo ponto extremo superior pertence a reta $y = 1$ e o ponto inferior pertence a reta $y = 0$. Estes segmentos particionam a faixa horizontal $[-\infty : \infty] \times [0 : 1]$ em $n+1$ regiões (veja Figura 14). Descreva um algoritmo de complexidade de tempo $O(n \log n)$ que contrói uma árvore de busca binária com os segmentos de S tal que a região contendo um dado ponto possa ser determinada em tempo $O(\log n)$. Também descreva o algoritmo de busca em detalhes.

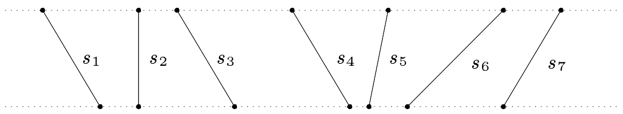


Figura 14. Segementos particionando a faixa $[-\infty : \infty] \times [0 : 1]$.

(2) (Exercício 35.2-2 de [6]) Dados dois segmentos de reta a e b que não se intersectam e são comparáveis na abscissa x , mostre como usando produto interno (i.e. as primitivas `Left`, ou `Lefton`, ou ..., ou o que você precisar) podemos decidir se $a >_x b$ em tempo $O(1)$.
[Note que no algoritmo `Intersecção-de-Segmentos` as operações `Insert, Delete, Above` e `Below` usam esta comparação.]

(3) (Exercício 35.2-3 de [6]) O professor Wellsmart sugeriu que modificássemos o algoritmo `Intersecção-de-Segmentos` de tal forma que ao invés do algoritmo parar logo após detectar uma intersecção o algoritmo imprime os segmentos que se intersectam e continua executando a próxima iteração do laço **para** (**for** *loop*). O professor chama o algoritmo resultante de `Imprime-Segmentos-Intersectantes` e afirma que esse algoritmo imprime todas as intersecções, da esquerda para a direita, a medida que elas vão ocorrendo. Mostre que o professor Wellsmart está duplamente enganado. Primeiro mostre um exemplo onde a primeira intersecção encontrada pelo algoritmo `Imprime-Segmentos-Intersectante` não é a mais à esquerda e depois mostre um exemplo onde o algoritmo não encontra todas as intersecções.

(4) (Exercício 35.2-4 de [6]) Mostre um algoritmo de complexidade de tempo $O(n \log n)$ que decide se um polígono de n vértices é simples.

(5) (Exercício 35.2-5 de [6]) Mostre um algoritmo de complexidade de tempo $O(n \log n)$ que decide se dois polígonos simples com um total de n vértices se intersectam.

(6) (Exercício 35.2-2 de [6]) Um *disco* (ou circulo) é a região do plano limitada por um circunferência. Um disco é dado através das coordenadas do seu centro e do valor do seu raio. Dois discos intersectam se eles tem algum ponto em comum. Descreva um algoritmo de complexidade de tempo $O(n \log n)$ que dados n discos decide se qualquer dois deles intersectam.

(7) (Exercício 2.11 de [7]) Dado um conjunto S de n circunferências no plano. Descreve um algoritmo por linha-de-varredura que determina todas as intersecções entre as circunferências. Como estamos tratando de circunferências aqui, e não discos, duas circunferências não se intersectam se um esta inteiramente no interior da região limitada pela outra. O seu algoritmo deve ter complexidade de tempo $O((n+I)\log n)$, onde I é o número de intersecções.

(8) (Exercício 35.2-7 de [6]) Dado um conjunto de n segmentos de reta contendo um total de k intersecções, mostre um algoritmo que devolve as I intersecções em tempo $O((n+I)\log n)$.
[Sugestão. Veja página 284 de [14] ou as notas de aula da disciplina.]

(9) (Exercício 2.14 de [7]) Dado um conjunto S de n segmentos de reta disjuntos no plano e um ponto p que não pertence a nenhum dos segmentosde S. Desejamos determinar todos os segmentos de S que são visíveis a partir de p, isto é, todos os segmentos de S que contém algum ponto q tal que o segmento aberto pq não intersecta nenhum dos segmentos de S. Descreva um algoritmo de complexidade de tempo $O(n \log n)$ para este problema. O seu algoritmo deve usar uma semireta com extremidade p e rotacionando ao redor de p (como os radares que vocês veêm nos fimes).

Referências

[1] I.J. Balaban, An optimal algorithm for finding segment intersections, *Proceedings of the 11th Annual ACM Symposium on Computational Geometry*, 1995, pp. 211–219.

[2] J.L. Bentley e T.A. Ottmann, Algorithms for reporting and counting geometric intersections, *IEEE Trans. Comput.* **C-28** (1979), 643–647.

[3] K.Q. Brown, Comments on "Algoritms for reporting and counting geometric intersections", *IEEE Trans. Comput.* **C-30** (1981), 147–148.

[4] B.M. Chazelle e H. Edelsbrunner, An optimal algorithm for intersecting line segments, *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, Oct. 1988 1988.

[5] K.L. Clarkson e P.W. Shor, Applicaions of random sampling in computational geoemtry, II, *Discrete and Computational Geometry* **4** (1989), 387–421.

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

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

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

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

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

[11] K. Mulmuley, A fast planar partition algorithm, I, *Proceedings of the 29th Annual IEEE Symposium on Fooundations of Computer Science*, 1988, pp. 580–589.

[12] J. O'Rourke, *Computational geometry in C*, Cambridge University Press, Cambridge, 1993, Second Edition, 1998.

[13] J. Pach e M. Sharir, On vertical visibility in arrangements of segments and the queue size in the bentley-ottman line sweeping algorithm, *SIAM Journal on Computing* **20** (1991), 460–470.

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

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