Notas de
Geometria Computacional

Luiz Henrique de Figueiredo
Paulo Cezar Pinto Carvalho

Instituto de Matemática Pura e Aplicada

versão preliminar para uso pessoal — não distribuir
texto atualizado em 7/3/2005

Prefácio

Estas são notas de aulas para o curso de Geometria Computacional ministrado anualmente no IMPA. Elas tiveram origem no livro escrito por nós para o 18° Colóquio Brasileiro de Matemática, em julho de 1991, e foram re-escritas de modo a refletir como o curso tem sido dado nos últimos dez anos.

Índice

1. O que é Geometria Computacional?

Vamos usar a seguinte definição:

> *Geometria Computacional é o estudo sistemático de algoritmos eficientes para problemas geométricos.*

Temos agora que explicar os termos usados nessa definição: O que são algoritmos? O que é eficiência? O que é um problema geométrico?

Vamos começar respondendo a última pergunta, não propriamente definindo o que é um problema geométrico, mas sim listando as suas principais características:

- A *entrada* ou os *dados* de um problema é um conjunto finito de objetos geométricos (pontos, retas, segmentos, triângulos, círculos, polígonos, etc.). Como *solução* do problema, devemos *calcular* algum número relacionado com a entrada (por exemplo, a área de um polígono), *construir* outros objetos geométricos (por exemplo, a interseção de dois polígonos), ou *decidir* se a entrada tem uma certa propriedade (por exemplo, se um polígono é convexo).
- Os objetos geométricos são definidos por *números reais*, através de coordenadas ou equações.
- Os objetos geométricos são *contínuos*, mas têm um descrição *discreta* — isso permite que eles sejam representados num computador. Por exemplo, um polígono é um região do plano, com um número infinito de pontos, mas que pode ser representada pela sua fronteira, que por sua vez pode ser representada pela sequência dos vértices.
- A solução de um problema geométrico envolve aspectos *geométricos* (localização e forma dos objetos) e aspectos *topológicos* (relações de adjacência e incidência entre os objetos). Sendo assim, um algoritmo geométrico envolve partes numéricas, correspondentes ao cálculo da geometria, e partes discretas, correspondentes ao cálculo da topologia. A parte geométrica geralmente é pequena mas essencial: ela vai determinar a parte topológica. Saber escolher como tratar a parte

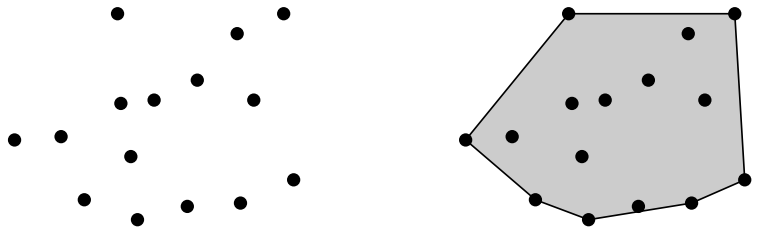
geométrica de forma simples, mas eficiente e robusta, é uma das chaves principais na resolução computacional de problemas geométricos.

Como veremos, essas características implicam em certas particularidades dos problemas geométricos que tornam a sua solução robusta uma tarefa difícil. Essas particularidades estão ausentes na maioria dos problemas e algoritmos estudados em ciência da computação.

A mistura de técnicas geométricas, numéricas e discretas, aliada à motivação proveniente de problemas práticos, dão à geometria computacional uma riqueza característica.

Exemplos. Vejamos alguns exemplos típicos de problemas geométricos. Nas figuras abaixo, a figura do lado esquerdo mostra os dados do problema e a figura do lado direito mostra a solução.

Fecho convexo



Interseção de polígonos

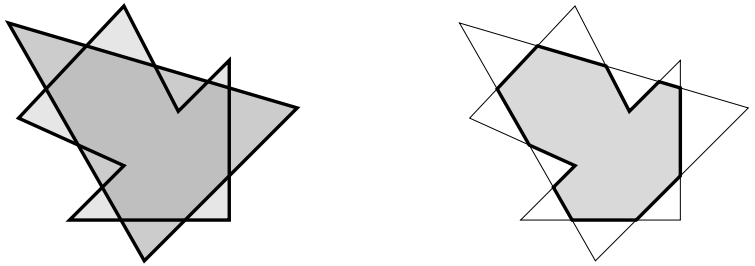
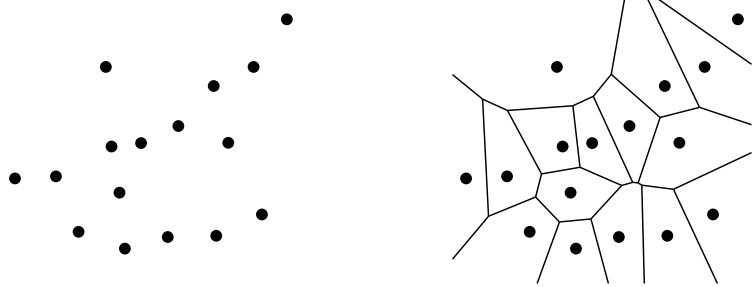
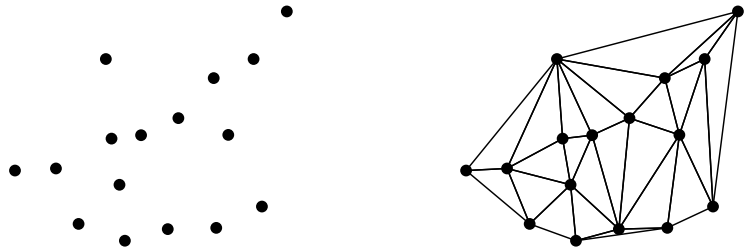


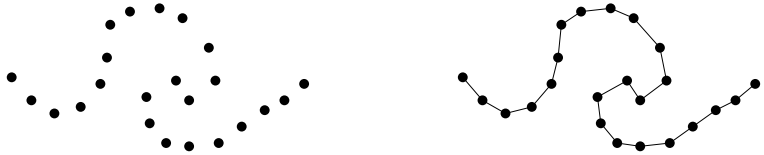
Diagrama de Voronoi



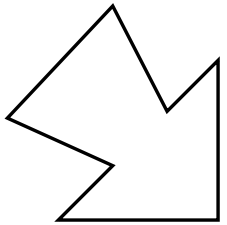
Triangulação



Reconstrução de curvas



Teste de convexidade



Não é convexo

Algoritmos. Um *algoritmo* é uma receita matemática de como resolver um problema. Um algoritmo descreve precisamente como combinar certas operações básicas para encontrar a solucão do problema. Precisamos de um *modelo computacional* para discutir precisamente que algoritmos são válidos, isto é, quais são as operações básicas e como elas podem ser combinadas, e também qual o *custo* de cada operação básica — isso vai permitir analisar a eficiênca dos algoritmos. Discutiremos no próximo capítulo o modelo teórico que vamos adotar: árvores de decisões algébricas.

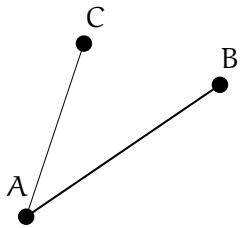
O *custo* de um algoritmo será medido pela soma dos custos das operações básicas que ele executa, ou seja, pela quantidade de recursos computacionais usados na sua execução, tipicamente quanto *tempo* ele leva para resolver um problema. (A quantidade de *memória* gasta pelo algoritmo também é uma medida importante de custo.) Veremos que não é possível nem desejável fazer uma análise detalhada do tempo necessário para resolver cada problema individualmente. Estudaremos a eficiência de um algoritmo estudando como o tempo de resolução do problema varia em função do tamanho da entrada, isto é, de quantos dados temos que processar (números de pontos, número de lados de um polígono, etc.). Além disso, na ausência de uma boa definição de caso médio, estudaremos a eficiência dos algoritmos fazendo uma *análise assintótica de pior caso*. Os algoritmos eficientes serão aqueles cujo custo cresce devagar com o tamanho da entrada.

Os algoritmos eficientes para problemas geométricos são fruto da combinação de teoremas matemáticos sobre a geometria do problema (frequentemente teoremas clássicos, conhecidos há séculos, mas também teoremas novos inspirados pela abordagem algorítmica), técnicas algorítmicas tradicionais e especiais, estruturas de dados adequadas (algumas bastantes sofisticadas), boa análise de desempenho, e primitivas geométricas simples e poderosas.

Em geral, basta usar um pequeno número de primitivas geométricas bem escolhidas; essas primitivas tendem a ser usadas em vários algoritmos. Um exemplo típico de primitiva geométrica, central em muitos dos algoritmos que vamos estudar, é o seguinte:

> *Dados três pontos no plano,* A, B *e* C, *decidir se o ponto* C *está à esquerda do segmento orientado* AB.

Esse predicado pode ser resolvido considerando o sinal do produto vetorial $AB \times AC$ e portanto pode ser implementado através da avaliação de um determinante.



Tipos de problemas. Podemos classificar informalmente os problemas abordados em geometria computacional em quatro tipos:

Seletivos: Nesses problemas queremos selecionar um subconjunto da entrada. Não temos que construir nenhum objeto geométrico novo, mas possivelmente temos que descobrir relações topológicas. Exemplos: fecho convexo, triangulação, árvore geradora mínima, simplificação de curvas e superfícies.

Construtivos: Nesses problemas temos que construir um ou mais objetos geométricos novos a partir da entrada, além de possivelmente relações topológicas envolvendo tanto objetos originais quanto objetos novos. Exemplos: interseção de polígonos, círculo mínimo, diagrama de Voronoi, geração de malhas, suavização de curvas e superfícies.

Decisão: Nesses problemas temos somente que responder "sim" ou "não" a uma pergunta. Não precisamos construir nada, nem novos objetos geométricos nem novas relações topológicas. Exemplo: Dentre um conjunto de segmentos de reta, há algum par que se cruza? (Não é necessário calcular *aonde* é o cruzamento, somente identificar *um* par que se cruza, se existir algum.) Outro exemplo: Dado um polígono, ele é convexo? Mais um: Dado um polígono e um ponto, o ponto está fora do polígono?

Consultas: Dado um conjunto fixo de objetos geométricos, queremos processá-lo previamente de modo a poder responder

eficientemente a consultas repetidas sobre ele. Exemplo: Dado um conjunto de pontos, a consulta é encontrar qual desses pontos está mais próximo de um ponto do plano. Outro exemplo: Dado um conjunto de retângulos, a consulta é listar todos os retângulos que intersectam um dado retângulo.

Aplicações. Os problemas tratados em geometria computacional ocorrem em muitas aplicações:

Computação Gráfica: Vários problemas de geometria computacional são motivados por problemas geométricos que aparecem em Computação Gráfica. Alguns exemplos: 1) Ao selecionarmos um objeto numa interface gráfica, devemos selecionar dentre todos os objetos desenhados na tela aquele que está mais próximo da posição do *mouse*. 2) Para desenhar de forma realista uma cena tridimensional, é necessário saber como os vários objetos se projetam na tela e tratar as suas oclusões. 3) Para fazer uma animação realista, é necessário detectar se há colisões entre os objetos que estão se movendo e o resto da cena.

Robótica: Um dos problemas fundamentais em robótica é o planejamento de movimento: o robô precisa analisar o seu ambiente e descobrir uma forma de se mover de um ponto ao outro sem colidir com os objetos no ambiente. Além disso, ele quer fazer isso da maneira mais eficiente possível, o que implica na necessidade de identificar o menor caminho viável entre os dois pontos.

Sistemas de informações geográficas: Esses sistemas lidam com enormes quantidades de dados geométricos para poder representar fielmente a geometria de estradas, rios, fronteiras, curvas de nível, áreas de vegetação, etc. Uma problema típico nessa área é saber que objetos geográficos estão perto de outros. Por exemplo, se um rio ameaça transbordar, quais as cidades e estradas que serão afetadas?

Circuitos integrados: Esses circuitos são compostos por dezenas de milhares de componentes eletrônicos que não podem se sobrepor, para evitar curto-circuitos. Durante o projeto desses circuitos é necessário identificar sobreposições eficientemente.

Bancos de dados: Uma consulta a bancos de dados com muitos campos numéricos é na verdade uma consulta multi-espacial. Por exemplo, se um banco de dados de uma companhia armazena idade, salário, altura, peso de cada funcionário, então cada funcionário é representado por um ponto em $\mathbf{R}^4$, e uma consulta ao banco de dados é uma busca em $\mathbf{R}^4$.

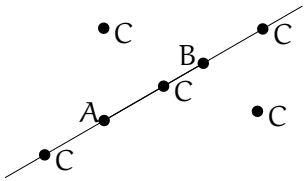
Configurações degeneradas. A principal particularidade dos problemas geométricos é a existência de *configurações degeneradas*, que ocorrem quando os objetos geométricos não estão em posição geral. Por exemplo: três pontos numa mesma reta, quatro pontos sobre um mesmo círculo, três retas passando pelo mesmo ponto. Essas configurações são chamadas de degeneradas porque elas ocorrem somente com probabilidade zero (o que não quer dizer que sejam impossíveis ou que não aconteçam na prática!). Qualquer algoritmo geométrico que se diga *robusto* deve estar preparado para lidar com casos especiais vindos de configurações degeneradas. Essa é uma fonte de complicações na implementação de algoritmos geométricos que infelizmente costuma ser ignorada tanto no desenvolvimento teórico quanto na prática.

Uma consequência importante da existência de configurações degeneradas é que os erros de arrendondamento da aritmética de ponto flutuante (usada pelos computadores para operar com os números reais que definem os objetos geométricos) passam a ser relevantes em situações *quase* degeneradas. Por exemplo, um ponto que esteja à esquerda de um segmento de reta e muito perto desse segmento pode ser confundido com um ponto que está à direita, e vice-versa. Como consequência, um algoritmo pode tomar decisões topológicas erradas ou inconsistentes. (Note que a topologia muda descontinuamente enquanto a geometria muda continuamente; portanto um pequeno erro na geometria pode acarretar um grande erro na topologia.)

Por outro lado, é importante frisar que de modo algum a aritmética de ponto flutuante é a responsável pelas configurações de-

generadas. Essas configurações existem mesmo que os objetos sejam definidos por números inteiros "pequenos", com os quais o computador pode operar sem erros. A aritmética de ponto flutuante somente exagera o problema da robustez de algoritmos geométricos. Felizmente, como veremos, é frequentemente possível isolar o uso de aritmética de ponto flutuante em um pequeno número de predicados primitivos, que podem ser implementados com cuidado por especialistas e usados pelo resto dos mortais.

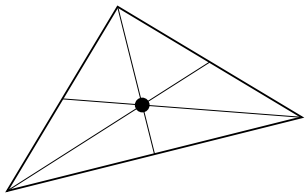
Por exemplo, mencionamos acima que um típico predicado geométrico é determinar se um ponto C está à esquerda de um segmento orientado AB. Genericamente, C pode estar à esquerda ou à direita de AB. Entretanto, isso não cobre todos os casos, pois



há configurações degeneradas: C pode estar sobre a reta AB — antes de A, em A, entre A e B, em B, ou depois de B. Uma implementação robusta desse predicado tem que tratar cuidadosamente todos esses casos.

Uma outra face das configurações degeneradas são os *invariantes geométricos*: mesmo objetos geométricos em posição geral podem gerar objetos secundários em configurações degeneradas. Muitos teoremas de geometria expressam fatos desse tipo. Eis aqui um exemplo muito conhecido:

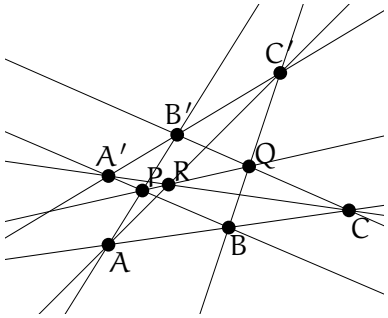
As três medianas de um triângulo são concorrentes.



(Uma mediana liga um vértice ao ponto médio do lado oposto.) Esse teorema permite definir o baricentro de um triângulo como o ponto comum às suas três medianas. Ainda que os vértices do triângulo estejam em posição geral no plano, as três medianas sempre são concorrentes (e portanto *não* estão em posição geral).

Um outro teorema desse tipo, bem menos conhecido e mais surpreendente, é o Teorema de Pappus:

Sejam L *e* L′ *duas retas no plano, e sejam* A, B, C *pontos de* L *e* A′, B′, C′ *pontos de* L′, *com* B *entre* A *e* C *e com* B′ *entre* A′ *e* C′. *Sejam* P = AB′ ∩ A′B, Q = AC′ ∩ A′C, *e* R = BC′ ∩ B′C. *Então,* P, Q *e* R *são colineares.*



Exemplo. Vejamos um exemplo típico de problema estudado em geometria computacional:

Dado um conjunto finito de pontos no plano, calcular o seu fecho convexo.

Historicamente, esse foi o primeiro problema a ser analisado completamente. Seguiremos esse caminho...

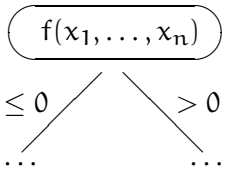
Para resolver esse problema, precisamos de vários peças:

- Definição matemática de fecho convexo
- Definição precisa do que é resolver esse problema algoritmicamente
- Teoremas que ajudem na solução algorítmica do problema
- Representação computacional da solução
- Estudo de soluções algorítmicas para o problema
- Identificação de primitivas geométricas adequadas
- Identificação e tratamento de casos especiais
- Análise de desempenho das soluções algorítmicas encontradas

2. Complexidade computacional

Para podermos discutir a eficência dos algoritmos, precisamos de um *modelo computacional*, que vai definir quais algoritmos são válidos e quais os custos das suas operações básicas. Um modelo computacional vai nos permitir inferir propriedades matemáticas dos algoritmos.

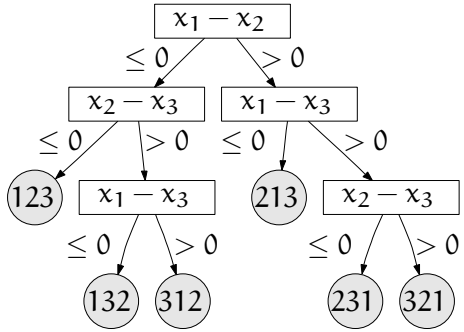
O modelo computacional que vamos adotar é o modelo de *árvores de decisões algébricas*. Nesse modelo, os algoritmos procedem avaliando expressões algébricas e tomando decisões baseadas no sinal desses valores. Mais precisamente, a entrada é uma sequência finita de números reais $x_1, x_2, \ldots, x_n$ (correspondentes às coordenadas e equações que representam os objetos geométricos) e um algoritmo é representado por uma árvore binária cujas folhas (nós terminais) contêm as possíveis respostas para o problema. Cada nó interno da árvore corresponde a um passo do algoritmo, no qual é avaliado um polinômio em $x_1, \ldots, x_n$.



De acordo com o sinal do resultado, o algoritmo prossegue para um dos descendentes do nó em questão: se o valor for positivo, o algoritmo vai para o nó da direita; caso contrário, o algoritmo vai para o nó da esquerda.

É importante ter em mente que o modelo descrito acima é somente isso, *um modelo*. Em outras palavras, ele é uma abstração de como os problemas são resolvidos nos computadores. Essa abstração tenta capturar uma parte essencial dos algoritmos, ao mesmo tempo que ignora muitos detalhes, de modo a nos fornecer um objeto matemático sobre o qual podemos provar teoremas. Além disso, esse não é o único modelo possível: existem muitos outros. Por outro lado, todos os modelos até hoje propostos são equivalentes, no sentido que um pode simular o outro completamente (como prevê a Tese de Church).

Como exemplo de como o modelo escolhido atua na prática, considere o problema de ordenar três números reais. A entrada é $x_1, x_2, x_3 \in \mathbf{R}$. A saída é uma permutação π do conjunto de índices $\{1,2,3\}$ tal que $x_{\pi(1)} \leq x_{\pi(2)} \leq x_{\pi(3)}$. Uma árvore de decisões algébricas para esse problema é ilustrada na página ao lado. Note que as folhas contêm todas as $6 = 3!$ permutações possíveis.



O custo (ou *complexidade*) de um algoritmo representado por uma árvore de decisões algébricas é medido pelo número de nós visitados na solução do problema, desde a raiz até uma folha (que contém a solução). Portanto, no modelo de árvores de decisões algébricas, as operações básicas — avaliar um polinômio — têm todas o mesmo custo (independente do grau do polinômio).

Note que o número de nós visitados pelo algoritmo depende da *instância* do problema. No exemplo acima, visitamos dois nós quando $x_1 \leq x_2 \leq x_3$ ou $x_2 \leq x_1 \leq x_3$, mas visitamos três nós nos outros casos. Dizemos então que esse algoritmo tem custo 3 *no pior caso*, isto é, no caso mais desfavorável.

Em geral, definimos a *complexidade* de um algoritmo como sendo o número máximo de nós visitados no pior caso, isto é, o seu custo na instância mais desfavorável. Mais precisamente, sejam P um problema que queremos resolver, A um algoritmo que resolve P e I uma instância de P. Denotamos por $T_A(I)$ o custo de A para resolver a instância I de P, isto é, o número de nós da árvore que representa A que são visitados na solução de P com entrada I. Como vimos no exemplo acima, $T_A(I)$ realmente varia com I, o que justifica a notação.

Um dos objetivos principais da análise de desempenho de algoritmos é poder comparar algoritmos para um mesmo problema, de modo a poder escolher o mais eficiente. Entretanto, é possível que dois algoritmos A e B para um mesmo problema sejam tais que A opera melhor sobre certas instâncias, enquanto B opera melhor sobre outras. Como então escolher entre A e B? Uma solução para esse impasse é expressar o custo de um algoritmo em

função da instância mais desfavorável. Em outras palavras, o pior caso dentre todas as instâncias de mesmo tamanho determina a complexidade T_A do algoritmo A, que passa então a ser uma função de n, o tamanho da entrada de P:

$$T_A(n) = \max_{|I|=n} T_A(I).$$

Há aqui uma sutileza: Estamos representando algoritmos como árvores binárias finitas. Essa representação somente se aplica para a solução de instâncias de mesmo tamanho: a árvore que ordena três números não serve para ordenar quatro números; precisamos de uma árvore maior, embora muito parecida com essa. Em outras palavras, um algoritmo que resolve um problema para instâncias de qualquer tamanho é na verdade representado por uma família de árvores, todas elas finitas, uma para cada tamanho de instância. Estamos interessados em como essas árvores crescem em função do tamanho n das instâncias. Para n fixo, o custo do algoritmo é medido pela *profundidade* da árvore correspondente a instâncias de tamanho n, isto é, pelo número máximo de nós visitados da raiz até uma folha.

A complexidade de pior caso, definida acima, pode não ser muito informativa na prática, pois o pior caso pode ser um caso especial muito raro. O ideal seria termos uma informação precisa sobre o desempenho no caso típico, chamado *caso médio*. Para isso, é necessário um modelo de distribuição probabilística das instâncias do problema. Entretanto, pode ser difícil encontrar um tal modelo: por exemplo, o que seria um polígono simples aleatório? Além disso, teríamos que fazer uma análise de desempenho para cada distribuição, pois distribuições diferentes dariam resultados diferentes. (Veremos que a complexidade dos algoritmos para fecho convexo para pontos aleatórios depende da região aonde eles são amostrados: por exemplo, retângulo e círculo dão resultados diferentes.)

Dadas essas dificuldades de tratar o case médio, continuamos com o estudo de complexidade de pior caso. Estamos interessados em estudar como $T_A(n)$ varia com n. Faremos uma *análise assintótica* de $T_A(n)$, isto é, uma análise de como $T_A(n)$ cresce quando n cresce. Isso permite resolver o impasse descrito acima:

Se temos dois algoritmos A e B competindo para a solução de um mesmo problema, e se conseguimos mostrar que $T_A(n)$ cresce mais rápido do que $T_B(n)$, então podemos dizer B que é mais eficiente do que A, pois, dado um orçamento computacional limitado, B vai permitir resolver problemas maiores. Em outras palavras, uma análise assintótica de complexidade de pior caso permite obter resultados que não dependem da tecnologia atual: mesmo que apareça um novo computador mil vezes mais rápido do que o atual, o algoritmo B ainda vai continuar sendo mais eficiente do que o algoritmo A para instâncias grandes.

A análise assintótica de $T_A(n)$ é feita estimando a ordem de grandeza do crescimento de $T_A(n)$ em função de n. As seguintes notações matemáticas expressam de maneira precisa esse tipo de estimativa:

Sejam $f, g: \mathbf{R} \to \mathbf{R}$ *duas funções positivas. Dizemos que:*

- g *é* $O(f)$ *quando existem* $N \in \mathbf{N}$ *e* $c \in \mathbf{R}$ *tais que* $g(n) \leq cf(n)$ *para todo* $n \geq N$.
- g *é* $\Omega(f)$ *quando existem* $N \in \mathbf{N}$ *e* $c \in \mathbf{R}$ *tais que* $g(n) \geq cf(n)$ *para todo* $n \geq N$.
- g *é* $\Theta(f)$ *quando* g *é* $O(f)$ *e* g *é* $\Omega(f)$.

Na prática escrevemos $g = O(f)$, $g = \Omega(f)$ e $g = \Theta(f)$, ainda que nesse caso "=" não seja uma relação simétrica (pelo contrário!).

É conveniente ter sempre em mente alguns exemplos típicos de crescimento assintótico. As funções abaixo estão listadas em ordem estritamente crescente, isto é, se f vem depois de g nessa lista, então $g = O(f)$ mas $g \neq \Theta(f)$:

$$1 \prec \log n \prec n^{1/k} \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec n^k \prec 2^n \prec n! \prec n^n.$$

Complexidade de problemas. Já sabemos como comparar algoritmos diferentes para um mesmo problema. Qual a melhor complexidade que se pode esperar de um algoritmo que resolva um dado problema? É possível comparar problemas diferentes? Existem problemas intrinsicamente mais difícieis do que outros?

Definimos a *complexidade* de um problema P como sendo o melhor que algum algoritmo que resolve P consegue fazer:

$$T_P(n) = \min_A T_A(n).$$

Note que esse mínimo é tomado sobre *todos* os algoritmos que resolvem P, sejam eles conhecidos ou não! Fica claro portanto que a análise da complexidade de um problema tem que ser feita através de teoremas sobre o problema P e os algoritmos que o resolvem, e não somente através da análise de desempenho de algoritmos particulares.

É claro que se A resolve P então $T_A = \Omega(T_P)$, isto é, nenhum algoritmo que resolva P pode ser mais rápido do que a complexidade de P. Em outras palavras, a existência de um algoritmo A que resolve P nos dá um limite superior, $T_P = O(T_A)$, mas nunca um limite inferior. Para obter um limite inferior é necessário analisar o problema P e descobrir propriedades de *todos* os algoritmos que resolvem P.

Dizemos que um algoritmo A é *ótimo* quando $T_A = O(T_P)$, ou seja, quando $T_A = \Theta(T_P)$ (ou equivalemente, $T_P = \Theta(T_A)$). Isso quer dizer que A consegue resolver P no melhor tempo possível. É importante saber T_P mesmo que não se conheçam algoritmos ótimos. Por outro lado, estimar a complexidade de um dado problema P é difícil. Como vimos, encontrar limites superiores, isto é, estimativas do tipo $O(f)$, é uma tarefa razoavelmente fácil, pois basta encontrar um algoritmo, mesmo ineficiente, que resolva P. Encontrar limites inferiores, isto é, estimativas do tipo $\Omega(f)$, é uma tarefa bem mais difícil, pois não é simples achar uma função f tal que $T_P = \Omega(f)$. Na verdade, frequentemente é simples achar limites inferiores triviais: por exemplo, um limite inferior para o problema de ordenação é $\Omega(n)$, porque, intuitivamente, qualquer algoritmo que consiga ordenar n números tem que testar todos os n números. O que se procura são limites inferiores não triviais; essa busca é difícil.

A prática de análise da complexidade de um problema P é uma espécie de corrida entre limites inferiores, obtidos por análise de P, e limites superiores, obtidos por análise de algoritmos que resolvam P, resultando num intervalo em torno de T_P:

$$\Omega(f) \preceq T_P \preceq O(g),$$

onde $\Omega(f)$ é o melhor limite inferior que se conhece para P e $O(g)$ é a complexidade do melhor algoritmo A que se conhece para P. O objetivo permanente da pesquisa sobre o problema P é diminuir esse intervalo, melhorando a análise de P, melhorando a análise de A, ou achando um algoritmo melhor do que A. Esse intervalo se fecha quando conseguimos encontrar um algoritmo ótimo para P. Nesse momento, podemos dizer que estudamos completamente a complexidade de P, pelo menos do ponto de vista da análise assintótica do pior caso.

Para o problema de ordenação, essa situação ideal se concretiza, como veremos agora.

Complexidade do problema de ordenação. Generalizando o que fizemos no exemplo de ordenação de três números, o problema de ordenação é formalmente o seguinte:

> *Dados* n *números reais* $x_1, x_2, \ldots, x_n \in \mathbf{R}$*, encontrar uma permutação* π *de* $\{1, 2, \ldots, n\}$ *tal que* $x_{\pi(1)} \leq x_{\pi(2)} \ldots \leq x_{\pi(n)}$.

Esse problema é muito importante em geometria computacional, pois além de ordenação ocorrer naturalmente na solução de muitos problemas, veremos que vários problemas podem ser analisados comparando-os com o problema ordenação. Sendo assim, vamos fazer uma análise completa da sua complexidade.

Começamos com um limite inferior não trivial:

$$T_{\text{ORDENAÇÃO}}(n) = \Omega(n \log n).$$

Em palavras:

> *No modelo de árvores de decisões algébricas, qualquer algoritmo que ordene* n *números leva tempo* $\Omega(n \log n)$.

A prova desse importante fato é surpreendentemente simples, pois temos um modelo computacional concreto com o qual podemos trabalhar. Vamos a ela.

Seja A um algoritmo que resolva o problema de ordenação. Então a árvore binária que representa A ao ordenar n números tem que ter pelo menos $n!$ folhas, pois esse é o número de soluções possíveis para o problema. Por outro lado, uma árvore binária de profundidade p tem no máximo 2^p folhas, como é fácil ver por indução. Concluímos então que $2^p \geq n!$. Logo $T_A(n) = p = \log_2(2^p) \geq \log_2 n!$ e portanto $T_A(n) = \Omega(\log n!)$.

Uma estimativa simples para $\log_2 n!$ quando $n \geq 4$ é

$$n! = 1 \cdot 2 \cdots n \geq 2 \cdot 2 \cdots 2 = 2^n.$$

Portanto $T_A(n) \geq \log_2 n! \geq \log_2 2^n = n$. Isso corresponde à nossa intuição de que para ordenar n números é necessário fazer pelo menos n comparações, pois todos os n números têm que ser considerados em algum momento. Entretanto, esse resultado simples só dá uma cota inferior trivial: $T_A(n) = \Omega(n)$. Para obter a cota inferior não trivial prometida acima, vamos estimar $\log_2 n!$ comparando $n!$ com n^n. Por um lado, temos que $n! \leq n^n$, pois

$$n! = 1 \cdot 2 \cdot 3 \cdots n \leq n \cdot n \cdot n \cdots n \leq n^n,$$

mas esse é um limite superior para $n!$ e precisamos de um limite inferior. Esse limite será dado pela estimativa $(n!)^2 \geq n^n$. De fato:

$$(n!)^2 = (1 \cdot 2 \cdot 3 \cdots (n-2) \cdot (n-1) \cdot n)(n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1).$$

Reagrupando os termos, obtemos

$$(n!)^2 = (1 \cdot n)(2 \cdot (n-1))(3 \cdot (n-2)) \cdots ((n-2) \cdot 3)((n-1) \cdot 2)(n \cdot 1) \geq n^n,$$

pois os todos termos são da forma ab com $ab \geq n$. (De fato, como $a + b = n + 1$, temos $a \geq 2$ e $b \geq n/2$ ou vice-versa, exceto pelo primeiro e pelo último termos, mas esses valem n.)

De $(n!)^2 \geq n^n$ concluímos então que

$$\log_2 n! \geq \frac{1}{2} \log_2(n^n) = \frac{1}{2} n \log_2 n$$

e portanto

$$T_A(n) = \Omega(\log n!) = \Omega(n \log n).$$

Note que, por causa das constantes embutidas na notação Ω, não precisamos nos preocupar com a base do logaritmo, já que todos as funções logaritmo são múltiplas umas das outras. □

Algoritmos para ordenação. Obtivemos acima um limite inferior não trivial $\Omega(n \log n)$ para o problema de ordenação. O limite trivial $\Omega(n)$ foi fácil de obter; o limite $\Omega(n \log n)$ precisou de uma análise mais detalhada, mas não muito mais complicada. Será que esse limite inferior $\Omega(n \log n)$ é o melhor possível? Não parece ser possível melhorar a análise que fizemos, pelo menos não na estimativa assintótica de $\log n!$, pois na verdade mostramos que $\log n! = \Theta(n \log n)$. Assim, se queremos melhorar o limite inferior para para o problema de ordenação, temos que fazer uma análise diferente da que fizemos, o que parece ser complicado. Portanto, vamos atacar na outra direção, tentando achar limites superiores.

Como discutimos, limites superiores para a complexidade de problemas vêm de algoritmos que resolvam esse problema. Que algoritmos existem para o problema de ordenação? Quais as suas complexidades? Qual o melhor algoritmo conhecido? Essas são as questões que vamos atacar agora.

Começamos com uma pergunta que deve ser feita em relação a qualquer problema que desejamos resolver num computador: Existe *alguma* solução algorítmica para o problema? Esse pergunta pode parecer trivial, mas é relevante porque nem todo problema tem solução algorítmica (o exemplo mais famoso é o problema da parada, o *halting problem*).

Algoritmo de força bruta. Para o problema de ordenação a resposta a essa pergunta é "sim", porque existe somente um número *finito* de soluções a se testar: as $n!$ permutações de $\{1, 2, \ldots, n\}$. Em outras palavras, um algoritmo óbvio para ordenar n números $x_1, x_2, \ldots, x_n$ é considerar todas permutações π de $\{1, 2, \ldots, n\}$ e testar se $x_{\pi(1)} \leq x_{\pi(2)} \leq \cdots \leq x_{\pi(n)}$. Pelo menos uma permutação vai passar nesse teste.

Cada um desses testes pode ser feito em tempo $O(n)$. Como temos $n!$ testes para fazer no pior caso, esse algoritmo leva tempo $O(n \cdot n!)$. É claro que esse algoritmo de força bruta só tem interesse teórico, pois ele leva tempo exponencial (lembre-se que $n! > 2^n$), mas pelo menos ele serve para dar um limite superior para o problema de ordenação, ainda que esse limite seja enorme! Temos portanto uma motivação para procurarmos algoritmos melhores.

Ordenação por seleção. Um algoritmo muito mais eficiente do que o algoritmo de força bruta é o algoritmo de ordenação por seleção. Como o nome sugere, esse algoritmo seleciona os números dados $x_1, x_2, \ldots, x_n$ em ordem crescente. Mais precisamente, no passo k o algoritmo seleciona $x_{\pi(k)}$, isto é, o número que ocupa a posição k na lista ordenada. Essa seleção é feita identificando o menor dos $n-k+1$ números ainda não selecionados. Para isso, temos que fazer $n-k$ comparações. O total de comparações é portanto

$$(n-1)+(n-2)+\cdots+2=\frac{n(n-1)}{2}-1=\Theta(n^2).$$

A análise que fizemos acima é atípica porque fomos capazes de calcular o número exato de operações básicas (nesse caso, comparações) gastas em qualquer instância de tamanho n. Além disso, esse número é independente da instância escolhida. No caso geral, não é possível calcular o número exato de operações básicas que um algoritmo executa para resolver cada instância de um problema; o melhor que se consegue fazer são limites superiores assintóticos.

De qualquer forma, conseguimos então um algoritmo não trivial para o problema de ordenação, cuja complexidade é muito menor do que a do algoritmo de força bruta. A situação no momento é então

$$\Omega(n\log n)\prec T_{\text{ORDENAÇÃO}}(n)\prec O(n^2).$$

É possível diminuir esse intervalo?

Ordenação por inserção. Tentemos um outro algoritmo simples: ordenação por inserção. A característica desse algoritmo é que no passo k já ordenamos $x_1, x_2, \ldots, x_k$, de modo que no passo n temos todos os números ordenados. Em outras palavras, esse algoritmo ordena incrementalmente. Do passo k para o passo $k+1$, comparamos x_{k+1} com $x_k, x_{k-1}, \ldots, x_1$ para saber aonde ele entra na ordem correta. Isso requer então no máximo k comparações. Portanto, o número total máximo de comparações é

$$1+2+\cdots+(n-1)=\frac{n(n-1)}{2}=\Theta(n^2).$$

Essa é a mesma complexidade do algoritmo de ordenação por seleção. Entretanto, há uma diferença importante: enquanto ordenação por seleção faz *sempre* um número quadrático de comparações, para qualquer instância do problema, ordenação por inserção faz *no máximo* um número quadrático de comparações; por exemplo, se os números já estão ordenados, ordenação por inserção faz somente $n-1$ comparações.

Em outras palavras, o desempenho de ordenação por seleção é sempre o mesmo, independente da instância, mas o desempenho de ordenação por inserção depende da instância: esse algoritmo se adapta à instância, gastando mais tempo quando ela estiver mais fora de ordem (o número exato de comparações depende do número de inversões de ordem presentes nos dados de entrada). Sendo assim, ordenação por inserção é recomendado para instâncias quase ordenadas. Por outro lado, nada disso afeta a complexidade de pior caso, que é o nosso objetivo principal. Desse ponto de vista, ordenação por seleção e ordenação por inserção têm a mesma complexidade: $\Theta(n^2)$. (Não estamos dizendo que o problema de ordenação é $\Theta(n^2)$, mas sim que esse dois algoritmos levam tempo $\Theta(n^2)$, isto é, levam tempo $O(n^2)$ para qualquer instância e há instâncias que levam tempo $\Omega(n^2)$.)

Recapitulando: tentamos achar um algoritmo de ordenação melhor do que ordenação por seleção, mas o algoritmo que encontramos tem a mesma complexidade, o que continua nos dando um limite superior $O(n^2)$ para o problema. Fica então a dúvida: será que é possível melhorar esse limite superior? Existem algoritmos melhores ou todos levam tempo pelo menos quadrático? Em outras palavras, será que a análise que fizemos do problema de ordenação, que nos deu o limite inferior $\Omega(n\log n)$, não é a melhor possível?

Ordenação por intercalação. Na verdade, existe um algoritmo melhor: ordenação por intercalação (*mergesort*). Esse algoritmo é mais sofisticado do que os que já vimos; ele usa um dos paradigmas fundamentais de construção de algoritmos: a obtenção da solução de um problema através da combinação das soluções de problemas de tamanho menor. Esse paradigma, chamado *dividir-para-conquistar*, é o segredo da eficiência de muitos algoritmos.

No algoritmo de ordenação por intercalação, para ordenar um conjunto de n números, dividimos esse conjunto em duas metades, ordenamos cada uma dessas metades e reunimos essas metades já ordenadas. Para ordenar cada uma das metades, aplicamos *recursivamente* o mesmo algoritmo.

O desempenho de qualquer algoritmo de divisão-e-conquista depende do desempenho das suas três etapas fundamentais:

separação: decompor a instância do problema em duas instâncias de tamanhos menores;

recursão: aplicar recursivamente o mesmo algoritmo para cada uma dessas instâncias;

combinação: combinar os resultados para obter a solução da instância original.

No caso do algoritmo de ordenação por intercalação, é facil ver que as etapas de separação e combinação podem ser feitas em tempo proporcional a n. Logo, a sua complexidade $T(n)$ satisfaz

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

onde c é uma constante. Para simplificar a análise, suponhamos que n é uma potência de 2, ou seja, $n = 2^p$. Então temos

$$T(n) = 2T(n/2) + cn,$$

Essa equação também se aplica a $n/2$:

$$T(n/2) = 2T(n/4) + cn/2,$$

o que fornece

$$T(n) = 4T(n/4) + 2cn.$$

Continuando esse processo, obtemos:

$$T(n) = 2^p T(n/2^p) + pcn = nT(1) + cn\log_2 n = O(n \log n).$$

Logo, o algoritmo de ordenação por intercalação é um algoritmo ótimo para o problema de ordenação! Portanto,

$$T_{\text{ORDENAÇÃO}}(n) = \Theta(n \log n),$$

pois mostramos por análise do problema que ele é $\Omega(n \log n)$ e por exibição e análise de um algoritmo que ele é $O(n \log n)$. Isso completa a nossa análise de complexidade do problema de ordenação.

Redução. A análise de complexidade do problema de ordenação que fizemos acima será muito útil no estudo da complexidade de problemas geométricos, começando com o problema de fecho convexo que vamos estudar a seguir. A ferramenta que permite re-usar análises de complexidade é a *redução*, que vamos explicar agora.

Sejam P_1 e P_2 dois problemas. Suponha que sabemos transformar instâncias de P_1 em instâncias de P_2 e soluções de P_2 em soluções de P_1. Nesse caso, dizemos que podemos *reduzir* P_1 a P_2, isto é, para resolver uma instância de P_1 basta resolver uma instância adequada de P_2. Essa situação pode ser resumida no seguinte diagrama:

$$\begin{array}{ccccc} & P_1 & & P_2 & \\ & I_1 & \longrightarrow & I_2 & \\ A_1 & \downarrow & & \downarrow & A_2 \\ & O_1 & \longleftarrow & O_2 & \end{array}$$

Em outras palavras, se temos um algoritmo $A_2 \colon I_2 \mapsto O_2$ que resolve P_2, então obtemos um algoritmo $A_1 \colon I_1 \mapsto I_2 \mapsto O_2 \mapsto O_1$ que resolve P_1.

Na situação acima, o que podemos dizer sobre as complexidades? Para simplificar a análise, vamos ignorar o tempo gasto para transformar as instâncias e as soluções, assumindo (como ocorre muito frequentemente na prática) que esse tempo é assintoticamente menor do que os tempos gastos por A_1 ou A_2. Se sabemos que $T_{A_2} = O(g)$, então $T_{A_1} = \Theta(T_{A_2}) = O(g)$, e portanto $T_{P_1} = O(g)$, pois existe um algoritmo A_1 que resolve P_1 em tempo $O(g)$. Em outras palavras, podemos transferir para P_1 quaisquer limites superiores que sejam conhecidos para P_2. O poder da redução está principalmente na recíproca: podemos transferir para P_2 quaisquer limites inferiores que sejam conhecidos para P_1. De fato, se sabemos que $T_{P_1} = \Omega(f)$, então $T_{A_1} = \Omega(f)$ e portanto $T_{A_2} = \Theta(T_{A_1}) = \Omega(f)$. Logo, podemos concluir que

$T_{P_2} = \Omega(f)$, pois A_2 é arbirário. Em outras palavras, se existisse um algoritmo A_2 para P_2 que fosse assintoticamente mais rápido do que f, então o algoritmo A_1, obtido a partir de A_2, violaria o limite inferior $T_{P_1} = \Omega(f)$.

Como mencionamos, veremos uma aplicação dessa técnica de redução de problemas quando fizermos a análise do problema de fecho convexo, que vamos estudar a seguir.

Exercícios.

1. Mostre que $n^2 + 1000n = O(n^2)$, encontrando explicitamente N e c como na definição. Mostre que podemos tomar $c = 2$ e $N = 1000$, $c = 101$ e $N = 10$, $c = 1001$ e $N = 1$.
2. Mostre que g é $O(f)$ se e somente f é $\Omega(g)$.
3. Mostre que g é $\Theta(f)$ se e somente f é $\Theta(g)$.
4. Mostre que g é $\Theta(f)$ se e somente se existem $N \in \mathbf{N}$ e $c_1, c_2 \in \mathbf{R}$ tal que $c_1 f(n) \leq g(n) \leq c_2 f(n)$ para todo $n \geq N$.
5. Mostre que $\log n! = \Theta(n \log n)$ usando os argumentos do texto. Note que não temos $n! = \Theta(n^n)$, pois a aproximação de Stirling diz que $n! = \Theta(n^n e^{-n} \sqrt{n})$ (o que também implica que $\log n! = \Theta(n \log n)$).
6. Mostre por indução que uma árvore binária de profundidade p tem no máximo 2^p folhas.
7. Qual o pior caso para o algoritmo de ordenação por força bruta?
8. Qual o pior caso para o algoritmo de ordenação por inserção?
9. A recorrência

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

pode ser resolvida explicitamente mesmo que n não seja uma potência de 2. Mostre que a solução é

$$T(n) = nT(1) + c(n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + n).$$

Conclua que $T(n) = \Theta(n \log n)$.

3. Fecho convexo

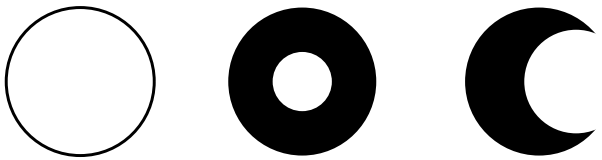
O primeiro problema geométrico que vamos estudar é o problema de calcular o fecho convexo de um conjunto finito de pontos no plano. Historicamente, esse foi o primeiro problema geométrico a ser completamente analisado. Além disso, esse problema tem uma rica relação com o problema de ordenação: a sua complexidade está intimamente ligada à de ordenação e muitos algoritmos para ordenação têm versões análogas para fecho convexo. Ainda assim, o problema de calcular o fecho convexo tem as suas particularidades, o que faz dele um problema ideal para uma introdução ao projeto e análise de algoritmos geométricos.

Como temos que fazer com todo problema geométrico que queremos resolver algoritmicamente, vamos começar definindo precisamente o que é o fecho convexo de um conjunto de pontos. A seguir, vamos estudar alguns resultados matemáticos na busca dos resultados relevantes à uma solução algorítmica.

Preliminares matemáticos. Um conjunto $K \subseteq \mathbf{R}^d$ é *convexo* quando, para quaisquer pontos $x, y \in K$, o segmento de reta xy ligando x e y em $\mathbf{R}^d$ está totalmente contido em K.

Exemplos de conjuntos convexos são: pontos, segmentos de reta, retas, discos, semiplanos, triângulos, etc.

Exemplos de conjuntos não convexos são: círculos, anéis, etc.



Conjuntos convexos são mais simples de lidar do que conjuntos quaisquer. Além disso, eles são frequentemente usados para

aproximar conjuntos mais complicados. Por exemplo, se estamos interessados em testar a sobreposição ou colisão de conjuntos no plano, um primeiro teste é testar isso para aproximações convexas simples desses conjuntos, como retângulos ou círculos. Se essas aproximações não se sobrepõem, então os conjuntos originais também não se sobrepõem (mas a recíproca não é verdadeira).

O *fecho convexo* de um conjunto $S \subseteq \mathbf{R}^d$ é o menor conjunto convexo de $\mathbf{R}^d$ que contém S, e é denotado por $\text{conv}(S)$. Dizemos *menor* no sentido que $\text{conv}(S)$ é um conjunto convexo contendo S e qualquer outro tal conjunto contém $\text{conv}(S)$. Mais precisamente, o fecho convexo de S é caracterizado por:

- $\text{conv}(S)$ é convexo;
- $S \subseteq \text{conv}(S)$;
- K convexo, $S \subseteq K \Rightarrow K \supseteq \text{conv}(S)$.

É fácil ver que a definição acima é equivalente a dizer que $\text{conv}(S)$ é a interseção de todos os conjuntos convexos que contêm S:

$$\text{conv}(S) = \bigcap_{\substack{K \text{ convexo} \\ K \supseteq S}} K.$$

De fato, seja C essa interseção. Então $C \supseteq S$, pois $S \subseteq K$ para todo K na interseção acima. Além disso, C é convexo, pois a interseção de qualquer família de convexos é convexa. Assim, C é um convexo que contém S. Logo, $C \supseteq \text{conv}(S)$, pela definição de $\text{conv}(S)$. Por outro lado, $\text{conv}(S)$ é um convexo que contém S e portanto aparece como K na interseção acima. Logo, temos $C \subseteq \text{conv}(S)$ e portanto $C = \text{conv}(S)$. □

Essas duas definições de fecho convexo são muito abstratas. Uma caracterização mais concreta usa a noção de combinação convexa. Dizemos que um ponto p é uma *combinação convexa* dos pontos $p_1, \ldots, p_n \in \mathbf{R}^d$ quando

$$p = \lambda_1 p_1 + \cdots + \lambda_n p_n, \quad \text{com} \quad \lambda_i \in \mathbf{R}, \quad \lambda_i \geq 0, \quad \sum_{i=1}^{n} \lambda_i = 1.$$

Assim, uma combinação convexa é um tipo especial de combinação linear. Como os coeficientes são todos positivos e somam 1, podemos ver uma combinação convexa como uma média ponderada dos pontos $p_1, \ldots, p_n$ com dados por $\lambda_1, \ldots, \lambda_n$.

Note que a noção de combinação convexa generaliza a noção de segmento de reta: os pontos no segmento de reta xy ligando x e y são combinações convexas de x e y, pois

$$xy = \{\, \lambda x + (1-\lambda) y : \lambda \in [0,1] \,\}.$$

Por outro lado, como veremos abaixo, toda combinação convexa pode ser obtida através de uma sequência de interpolações lineares desse tipo. Portanto, não é surpresa que a noção de combinação convexa esteja intimamente ligada à noção de conjunto convexo. É o que veremos a seguir.

Dizemos que um conjunto $K \subseteq \mathbf{R}^d$ é *fechado por combinações convexas* quando K contém todas as combinações convexas de pontos de K. Essa propriedade caracteriza os conjuntos convexos:

> *Um conjunto é convexo se e somente se ele é fechado por combinações convexas.*

Por um lado, como vimos acima, os pontos de um segmento de reta são combinações convexas dos seus extremos. Logo, se um conjunto é fechado por combinações convexas, então ele é convexo. Por outro lado, se temos $p_1, \ldots, p_n$ pontos de um conjunto convexo K e p uma combinação convexa desses pontos, digamos $p = \lambda_1 p_1 + \cdots + \lambda_n p_n$, então $p \in K$. A prova é por indução em n: Se $n = 1$, temos $p = \lambda_1 p_1 = p_1 \in K$. Para passar de n para $n+1$, seja $p = \lambda_1 p_1 + \cdots + \lambda_n p_n + \lambda_{n+1} p_{n+1}$ uma combinação convexa de $n+1$ pontos de K. Não podemos aplicar a hipótese de indução aos n primeiros termos de p porque $\sum_{i=1}^{n} \lambda_i \neq 1$, mas isso é fácil de corrigir tomando $q = \sum_{i=1}^{n} \mu_i p_i$, onde $\mu_i = \lambda_i / \mu$ e $\mu = \sum_{i=1}^{n} \lambda_i \neq 0$. (Se $\mu = 0$, então $\lambda_{n+1} = 1$ e $p = p_{n+1} \in K$.) Então $\mu_i \geq 0$ e $\sum_{i=1}^{n} \mu_i = 1$. Assim, q é uma combinação convexa de n pontos de K e portanto, por hipótese de indução, $q \in K$. Por outro lado, temos $p = \mu q + \lambda_{n+1} p_{n+1}$, com $\mu \geq 0$ e $\mu + \lambda_{n+1} = \sum_{i=1}^{n+1} \lambda_i = 1$. Em outras palavras, $p \in q p_{n+1} \subseteq K$, completando a prova por indução. □

Desse resultado, segue a seguinte caracterização de fecho convexo:

O fecho convexo de um conjunto S é o conjunto das combinações convexas de pontos de S.

De fato, seja C o conjunto das combinações convexas de pontos de S:

$$C = \{\, \lambda_1 p_1 + \cdots + \lambda_n p_n : \; p_i \in S,\; \lambda_i \in \mathbf{R},\; \lambda_i \geq 0,\; \sum_{i=1}^{n} \lambda_i = 1 \,\}.$$

Como $S \subseteq \mathrm{conv}(S)$ e, como vimos, conjuntos convexos são fechados por combinações convexas, temos $C \subseteq \mathrm{conv}(S)$. Por outro lado, temos $C \supseteq \mathrm{conv}(S)$, pois $S \subseteq C$ e é fácil ver que C é convexo. De fato, sejam $p, q \in C$. Queremos mostrar que $pq \subseteq C$. Adicionando termos com coeficientes nulos, se for necessário, podemos assumir que p e q são combinações convexas dos mesmos pontos de S:

$$\begin{aligned} p &= \lambda_1 p_1 + \cdots + \lambda_n p_n, \\ q &= \mu_1 p_1 + \cdots + \mu_n p_n. \end{aligned}$$

Então, para qualquer $\tau \in [0,1]$, temos

$$\tau p + (1-\tau) q = \tau_1 p_1 + \cdots + \tau_n p_n,$$

onde $\tau_i = \tau\lambda_i + (1-\tau)\mu_i \geq 0$ e ainda

$$\sum_{i=1}^{n} \tau_i = \tau \sum_{i=1}^{n} \lambda_i + (1-\tau) \sum_{i=1}^{n} \mu_i = \tau \cdot 1 + (1-\tau) \cdot 1 = 1.$$

Portanto, $\tau p + (1-\tau) q \in C$. Logo, $pq \subseteq C$. □

Fecho convexo de um conjunto finito. A caracterização de $\mathrm{conv}(S)$ como o conjunto das combinações convexas de pontos de S é bem mais concreta do que a definição abstrata em termos do menor conjunto convexo que contém S, mas ainda não é suficiente para podermos pensar em algoritmos para calcular $\mathrm{conv}(S)$. De

fato, mesmo que S seja finito, $\mathrm{conv}(S)$ ainda é um conjunto infinito se S tem pelo menos dois elementos, já que o segmento de reta que os liga é um conjunto infinito contido em S.

Precisamos de uma caracterização de $\mathrm{conv}(S)$ para S finito que seja *discreta*, isto é, representável num computador. Vamos usar a caracterização abaixo, que é muito intuitiva mas não trivial de provar:

O fecho convexo de um conjunto finito de pontos no plano é um polígono convexo cujos vértices são pontos do conjunto dado.

Sendo assim, para calcular $\mathrm{conv}(S)$ num computador basta identificar os pontos de S que são vértices de $\mathrm{conv}(S)$ e listá-los na ordem em que eles aparecem na fronteira de $\mathrm{conv}(S)$.

Vamos agora dar uma prova da caracterização acima, listando uma série de fatos elementares. Como essa caracterização descreve $\mathrm{conv}(S)$ em termos da sua fronteira, somos levados a considerar a fronteira de um conjunto convexo e os pontos de S que estão na fronteira de $\mathrm{conv}(S)$.

Começamos com a noção de ponto extremo, que se aplica a qualquer conjunto convexo K. Os pontos extremos serão os candidatos a vértices de $\mathrm{conv}(S)$. Dizemos que $p \in K$ é um *ponto extremo* de K quando p não pode ser escrito como combinação convexa não trivial de pontos de K. Em outras palavras,

$$p = \lambda_1 p_1 + \cdots + \lambda_n p_n \;\Rightarrow\; p = p_i \text{ para algum } i.$$

É fácil ver que basta tomar $n = 2$ nessa definição, isto é, um ponto é extremo de K se e somente se ele não está no interior de um segmento de reta ligando dois pontos de K.

O nosso primeiro passo na caracterização de $\mathrm{conv}(S)$ como um polígono convexo com vértices em S é o seguinte:

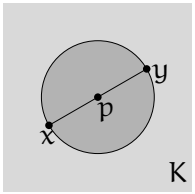
Os pontos extremos de $\mathrm{conv}(S)$ *estão em* S.

De fato, seja p ponto extremo de $\mathrm{conv}(S)$. Como $\mathrm{conv}(S)$ é o conjunto de combinações convexas de pontos de S, o ponto p é uma combinação convexa de pontos de S. Essa combinação tem que

ser trivial, pois p é extremo e $S \subseteq \text{conv}(S)$. Isso quer dizer que $p \in S$. □

O próximo passo diz que os pontos extremos ocorrem somente na fronteira:

Seja K *um conjunto convexo. Então os pontos extremos de* K *estão na fronteira de* K.

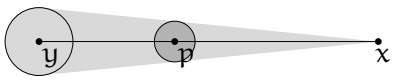


Vamos mostrar a contra-positiva: nenhum ponto interior pode ser ponto extremo. Seja p um ponto do interior de K. Então existe um círculo centrado em p e totalmente contido em K. Seja xy um diâmetro qualquer desse círculo. Então p é o ponto médio de xy e $x, y \in K$, isto é, p é uma combinação convexa não trivial de pontos de K. Portanto, p não é ponto extremo de K. □

O próximo passo diz que a fronteira é formada por pontos extremos e segmentos de reta ligando pontos extremos:

Se um ponto está na fronteira de K*, mas não é um ponto extremo, então ele está no interior de um segmento de reta definido por pontos extremos de* K.

Seja p um ponto na fronteira de K que não é ponto extremo. Então p está no interior de um segmento xy determinado por dois pontos $x, y \in K$. Se um desses pontos, digamos y, estiver no interior, então existe um círculo centrado em y e totalmente contido em K. Mas então esse círculo e o ponto x definem um "cone" totalmente contido em K. Dentro desse "cone" existe um círculo centrado em p e totalmente contido em K. Logo p está no interior de K. □



Resumindo o que temos até o momento: a fronteira de um conjunto convexo K é formada por pontos extremos e segmentos de reta. Quando K é limitado, esses segmentos de reta são limitados e portanto seus extremos são pontos extremos. Isso mostra que, quando S é finito, o fecho convexo de S é realmente um polígono convexo cujos vértices são pontos de S, como esperávamos. □

Assim, resolver computacionalmente o problema de achar o fecho convexo de um conjunto finito S de pontos no plano significa identificar os pontos extremos de $\text{conv}(S)$ dentre os pontos de S e listá-los na ordem circular em que eles aparecem na fronteira. Note como naturalmente aparece a ligação entre o problema do fecho convexo e o problema de ordenação.

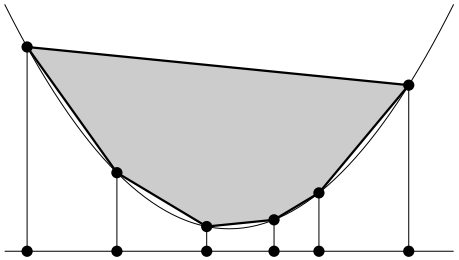
Complexidade. Antes de descrever algoritmos que resolvem o problema de fecho convexo, vamos usar a técnica de redução de problemas para concluir um limite inferior não trivial para esse problema.

Vamos reduzir o problema de ordenação ao problema de fecho convexo. Feito isso, vamos poder transferir para FECHO CONVEXO o limite inferior $\Omega(n \log n)$ estabelecido para ORDENAÇÃO.

Para reduzir ORDENAÇÃO a FECHO CONVEXO, temos que mostrar como converter qualquer instância de ORDENAÇÃO para uma instância de FECHO CONVEXO cuja solução pode ser convertida numa solução de problema de ordenação original.

Sejam então dados n números reais que queremos ordenar: $x_1, \ldots, x_n \in \mathbf{R}$. Temos transformar essa instância I_1 do problema P_1 de ORDENAÇÃO numa instância I_2 do problema P_2 de FECHO CONVEXO. Não podemos fazer uma transformação arbitrária: temos que escolher I_2 cuidadosamente, de forma a poder controlar a resposta O_2, pois é a partir dela que temos que obter a resposta O_1 do problema de ordenação que queremos resolver.

Em outras palavras, temos que escolher cuidadosamente os pontos $p_1, \ldots, p_n \in \mathbf{R}^2$ que formam I_2 de forma que o seu fecho convexo contenha a informação da ordem de $x_1, \ldots, x_n$. Como não podemos perder nenhum ponto, temos que escolher I_2 de modo que *todos* os pontos sejam vértices do fecho convexo de $\{p_1, \ldots, p_n\}$. Além disso, a ordem na qual eles ocorrem na fronteira deve refletir — de algum modo simples — a ordem na qual $x_1, \ldots, x_n$ ocorrem na reta. Esse é o ponto principal, claro.



Uma transformação simples que satisfaz essas condições é tomar $p_i = (x_i, x_i^2)$. Esses pontos estão todos sobre a parábola $y = x^2$. A região $y \geq x^2$ é uma região convexa, cuja fronteira é a parábola. Sendo assim, todos os pontos p_i são vértices do fecho convexo de I_2. Além disso, a ordem na qual eles ocorrem na fronteira, quando iniciada no ponto mais à esquerda, reflete exatamente a ordem na qual $x_1, \ldots, x_n$ devem ser listados.

Mais precisamente, seja A_2 um algoritmo que resolve o problema de fecho convexo no plano. Então a saída O_2 de A_2 com entrada I_2 (os pontos na parábola) é um polígono convexo com n vértices $v_1, \ldots, v_n$. Cada um desses vértices é um dos pontos originais; o algoritmo A_2 tem que dizer qual ponto corresponde a cada vértice. Porém, não sabemos como A_2 escolheu o vértice v_1 para começar a lista. Procedemos então do seguinte modo: Começando em v_1, visitamos sucessivamente os vértices de forma circular até encontrar um vértice v_k que está à esquerda de v_{i-1}. Nesse momento, teremos identificado o vértice v_k mais à esquerda. Começando em v_k, visitamos sucessivamente os vértices, listando a coordenada x dos vértices visitados, até retornar a v_k. A lista produzida é a resposta O_1 do problema original de ordenação.

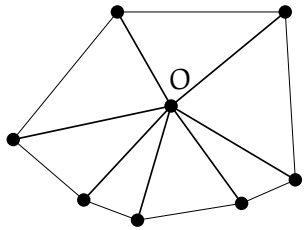
Assim, ORDENAÇÃO se reduz a FECHO CONVEXO. Como vimos no capítulo anterior, essa redução permite usar o limite inferior $\Omega(n \log n)$ estabelecido para ORDENAÇÃO para estabelecer um limite inferior para FECHO CONVEXO:

$$T_{\text{FECHO CONVEXO}}(n) = \Omega(n \log n).$$

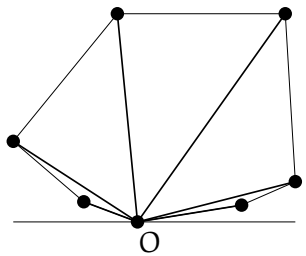
Esse é um limite inferior não trivial para FECHO CONVEXO. Note, entretanto, que não sabemos se esse é o *melhor* limite inferior possível para o problema. Pode ser que FECHO CONVEXO seja ainda mais difícil que ORDENAÇÃO. A redução acima mostra somente que FECHO CONVEXO é pelo menos tão difícil quanto ORDENAÇÃO.

Algoritmos. Já sabemos um limite inferior não trivial para FECHO CONVEXO. Não é claro como melhorar esse limite inferior. Tentemos então achar algoritmos que resolvam o problema, de modo a estabelecer limites superiores para o problema e saber quão longe estamos desse limite inferior não trivial.

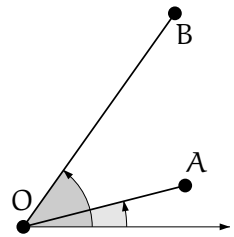
Como vimos, resolver computacionalmente o problema de achar o fecho convexo de um conjunto finito S de pontos no plano significa identificar os pontos extremos dentre os pontos de S e listá-los na ordem circular em que eles aparecem na fronteira de $\text{conv}(S)$. Essa ordenação é simples, uma vez que conheçamos os pontos extremos, pois eles estão ordenados *angularmente* em relação a qualquer ponto de $\text{conv}(S)$. Sabemos que podemos ordenar n números em tempo ótimo $\Theta(n \log n)$. Mas como fazer para ordenar *ângulos*? Felizmente, não é preciso calcular esses ângulos, somente compará-los. Essa comparação é simples se escolhermos adequadamente o ponto de referência $O \in \text{conv}(S)$.



Como mencionamos, qualquer ponto de $\text{conv}(S)$ serve como ponto de referência para a ordenação angular. Por exemplo, podemos pegar um dos pontos de S ou o baricentro de S. Entretanto, nem todos os pontos de $\text{conv}(S)$ são igualmente bons para a tarefa de ordenação angular. Uma escolha que simplifica essa ordenação é escolher o ponto mais baixo, isto é, o ponto O de S que tem menor coordenada y. (Em caso de empate, escolhemos o ponto que tiver menor coordenada x.)



Esse ponto O é certamente um ponto extremo de $\text{conv}(S)$, mas o que é importante para a nossa tarefa de ordenação angular é que podemos comparar angularmente dois vértices A e B de $\text{conv}(S)$ simplesmente localizando B em relação a OA. Mais precisamente, o ângulo que OB faz com a semireta horizontal



que passa por O é maior do que o ângulo que OA faz com essa semireta se e somente se B está à esquerda de OA. Note que isso não seria verdade se o ponto O fosse um ponto interior de $\text{conv}(S)$. Note ainda que OA e OB não podem ser colineares pois A, B e O são pontos extremos.

Sendo assim, para encontrar o fecho convexo de S resta então identificar os seus pontos extremos, pois já sabemos como ordená-los na ordem certa.

Não é claro como formular um critério direto para identificar pontos extremos. Porém, podemos identificar os pontos que *não* são pontos extremos, usando a seguinte propriedade:

> *Se* $p \in S$ *não é um ponto extremo de* $\text{conv}(S)$*, então existem* $a, b, c \in S \setminus \{p\}$ *tal que* p *está no triângulo* abc.

Em outras palavras, um ponto que não é extremo está em algum triângulo definido por pontos de S mas não é nenhum dos vértices. (Ele pode estar no interior do triângulo ou no interior de alguma aresta.) Ou ainda: um ponto não é extremo se e somente se ele é combinação convexa não trivial de *três* pontos de S.

Essa caracterização dos pontos que não são extremos nos dá um algoritmo para identificar pontos extremos: Considere todos os triângulos definidos por pontos de S. Para cada triângulo, elimine os pontos que estão no seu interior ou no interior de alguma das suas arestas. Os pontos que sobrarem são os pontos extremos de $\text{conv}(S)$.

Para saber se um ponto p está no interior de um triângulo abc, basta classificar p em relação a cada uma das suas arestas. Assumindo que o triângulo abc está orientado positivamente, temos que p está no interior de abc se e somente se p está à esquerda de ab, bc, e ca.

Finalmente temos um algoritmo completo que calcula o fecho convexo $\text{conv}(S)$ de um conjunto finito S de pontos do plano:

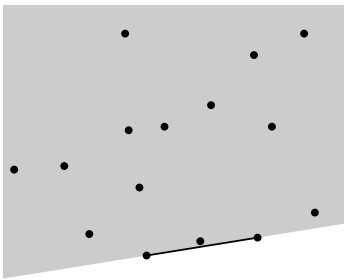
1. Identifique os pontos extremos usando o critério acima.
2. Ordene os pontos extremos angularmente em relação ao ponto mais baixo em S.

Qual o custo desse algoritmo? No passo 1 temos que considerar $\binom{n}{3}$ triângulos e testar n pontos para cada triângulo. Logo esse passo leva tempo $n\binom{n}{3} = O(n^4)$. O passo 2 pode ser feito em tempo $O(n \log n)$. Assim, esse nosso primeiro algoritmo leva tempo $O(n^4) + O(n \log n) = O(n^4)$. Ainda estamos muito longe do limite inferior não trivial $\Omega(n \log n)$ que estabelecemos para o problema!

Algoritmo 2 (arestas). O nosso primeiro algoritmo concentrou a atenção nos vértices de $\text{conv}(S)$, motivado por uma caracterização algorítmica dos pontos extremos. Será que mudar o foco da atenção para as arestas resulta num algoritmo melhor? Um fato geométrico relevante aqui é o seguinte:

> *As arestas de* $\text{conv}(S)$ *são definidas por pontos de* S *e estão em retas suporte de* S.

Mais precisamente, pq é uma aresta de $\text{conv}(S)$ se e somente se todos os pontos de S estão de um mesmo lado de pq, podendo estar sobre a aresta pq. Orientando a fronteira de $\text{conv}(S)$ positivamente, temos a seguinte caracterização das arestas de $\text{conv}(S)$:



> *Sejam* $p, q \in S$*. Então* pq *é uma aresta de* $\text{conv}(S)$ *orientada positivamente se e somente se todos os pontos de* S *estão à esquerda de* pq *ou sobre* pq.

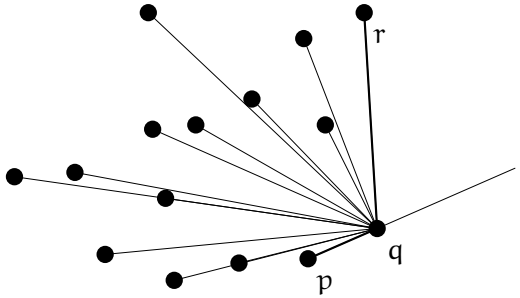
Essa caracterização nos dá um novo algoritmo:

1. Identifique as arestas usando o critério acima.
2. Monte a fronteira a partir das arestas.

Qual o custo desse algoritmo? No passo 1 temos que considerar $\binom{n}{2}$ arestas e classificar n pontos em relação a cada aresta. Logo esse passo leva tempo $n\binom{n}{2} = O(n^3)$. Como fazer o passo 2? É fácil fazer em tempo $O(n^2)$, montando a fronteira incrementalmente a partir de uma aresta qualquer: a cada passo temos um trecho completo e orientado da fronteira e procuramos a próxima aresta. Como o passo 1 já leva tempo $O(n^3)$, não vale a pena melhorar o passo 2, mas na verdade ele pode ser feito em tempo $O(n)$.

Algoritmo 3 (Jarvis). Melhoramos o limite superior, mas ainda estamos longe do limite inferior. Será possível melhorar o limite superior encontrando um algoritmo melhor? O modo como um ser humano resolve o problema de fecho convexo não é identificando as arestas individualmente e depois montando a fronteira: é identificando a fronteira na ordem correta! Essa é a idéia do algoritmo de Jarvis (1973) que vamos descrever agora. Esse algoritmo se baseia na seguinte caracterização geométrica:

> *Se* pq *é uma aresta de* $\mathrm{conv}(S)$ *orientada positivamente, então a próxima aresta é* qr*, onde* r *é o ponto de* S *tal que* qr *define o menor ângulo com a semireta* pq.



O algoritmo de Jarvis começa encontrando uma aresta v_1v_2 de $\mathrm{conv}(S)$. Para isso, ele toma v_1 como o ponto mais baixo e mais à esquerda em S. Como vimos, v_1 é um ponto extremo de $\mathrm{conv}(S)$ e todos os outros pontos de S estão acima ou sobre a semireta horizontal que parte de v_1 para a direita. O ponto v_2 é aquele que define o menor ângulo com essa semireta. Também como já vimos, o ponto v_2 pode ser encontrado comparando as posições dos pontos de S em relação à semireta: r define um ângulo maior com a semireta do que s se e somente se r está à esquerda de v_1s.

Tendo encontrado a aresta inicial v_1v_2, encontramos a próxima aresta v_2v_3 achando o ponto v_3 tal que v_2v_3 define o menor ângulo com a semireta v_1v_2. Continuamos esse processo até voltarmos a v_1.

Quanto tempo leva esse algoritmo? Para encontrar cada aresta, temos que testar $O(n)$ pontos. Como temos no máximo n arestas no fecho convexo, o algoritmo leva tempo $O(n^2)$, sendo portanto o melhor algoritmo até o momento.

Embora esse seja um bom resultado — o melhor até agora — vamos analisar o algoritmo de Jarvis com mais cuidado. Realmente, temos que testar $O(n)$ pontos para encontrar cada aresta, mas temos sempre n arestas? Nem sempre, depende da instância. Seja então h o número de arestas em $\mathrm{conv}(S)$. Claro que temos $h = \Theta(n)$, pois no pior caso temos $h = n$ (quando todos os pontos de S são pontos extremos; por exemplo, pontos sobre um círculo). Mas podemos ter $h = 2$, quando os pontos são todos colineares, ou $h = 3$, quando os pontos estão dentro de um triângulo. Portanto, uma análise mais fina do algoritmo de Jarvis nos diz que ele leva tempo $O(nh)$, pois ele leva tempo $O(n)$ para encontrar cada uma das h arestas.

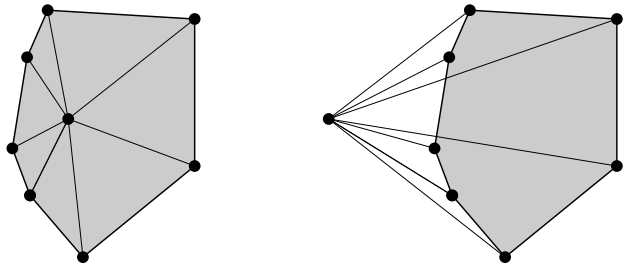
Note que essa complexidade agora está sendo dada não só em função do tamanho n da entrada, mas também em função do tamanho h da saída. Essa é uma boa atitude, pois o tamanho da solução do problema de fecho convexo depende da instância. É natural então medir a complexidade desse problema e de seus algoritmos usando também o tamanho da saída. (Note aqui uma diferença fundamental entre ORDENAÇÃO e FECHO CONVEXO: a saída de ORDENAÇÃO tem sempre tamanho n.)

É possível mostrar que $h = \Theta(n^{1/3})$ quando os pontos são escolhidos aleatoriamente dentro de um círculo e que $h = \Theta(\log n)$ quando os pontos são escolhidos aleatoriamente dentro de um polígono convexo. Note que nesse último caso, o algoritmo de Jarvis é ótimo! (Mas ele continua sendo $\Theta(n^2)$ no pior caso.)

Algoritmo 4. O algoritmo de Jarvis constrói $\mathrm{conv}(S)$ incrementalmente, uma aresta de cada vez. Veremos agora um outro algoritmo incremental, motivado pela seguinte pergunta: o que acontece se consideramos um ponto de cada vez?

Mais precisamente, seja $S = \{p_1, \ldots, p_n\}$. Qual a relação entre $K = \mathrm{conv}(S)$ e $K' = \mathrm{conv}(S')$, onde $S' = S \setminus \{p_n\} = \{p_1, \ldots, p_{n-1}\}$? Usando a caracterização de $\mathrm{conv}(S)$ como a união de todos os triângulos com vértices em S, concluímos que K é a união de K' com todos os triângulos uvp_n com u, v vértices de K', já que K' é o fecho convexo dos seus vértices. Se $p_n \in K'$, esses triângulos estão todos em K' e portanto $K = K'$. Se $p_n \notin K$, temos que substituir

um trecho da fronteira de K' por duas arestas passando por p_n. Supondo que a fronteira de K' está orientada positivamente, então as arestas que temos que remover são as arestas uv para as quais v está à esquerda ou sobre up_n.



Essa construção nos dá um algoritmo incremental: Começamos com $K_2 = \text{conv}(S_2)$, onde $S_2 = \{p_1, p_2\}$. Tendo construído $K_k = \text{conv}(S_k)$, onde $S_k = \{p_1, \ldots, p_k\}$, passamos para K_{k+1} usando a construção acima. Depois de $n-1$ passos, teremos $K_n = \text{conv}(S_n) = \text{conv}(S)$, que é o que queremos.

Quanto tempo leva esse algoritmo? A cada passo, podemos decidir se $p_{k+1} \in K_k$ em tempo $O(h_k)$, onde h_k é o número de arestas em K_k. Também levamos tempo $O(h_k)$ para obter K_{k+1} a partir de K_k, quando $p_{k+1} \notin K_k$. O tempo total que esse algoritmo leva é $h_2 + h_3 + \cdots + h_{n-1}$. No pior caso, temos $h_k = k$. O pior caso ocorre quando todos os pontos de S são pontos extremos de $\text{conv}(S)$ (por exemplo, pontos sobre um círculo). Logo o tempo total no pior caso é $2 + 3 + \cdots + (n-1) = \binom{n}{2} - 1 = O(n^2)$.

Assim, esse algoritmo incremental não é melhor do que o algoritmo de Jarvis e parece até ser mais complicado. Entretanto, como veremos a seguir, é possível modificá-lo para simplificar as tarefas custosas: decidir se $p_{k+1} \in K_k$ e mudar a fronteira de K_k para incluir p_{k+1}, quando $p_{k+1} \notin K_k$.

Algoritmo 5. Em vez de ter que decidir se $p_{k+1} \in K_k$, vamos forçar que esse caso nunca aconteça: vamos ordenar os pontos *lexicograficamente*, isto é, usando as suas coordenadas x e as coordenadas y em caso de empate. É como os pontos fossem palavras de duas letras que vamos ordernar em ordem alfabética, como num dicionário.

Se os pontos estão ordenados lexicograficamente, então temos sempre $p_{k+1} \notin K_k$, porque existe uma reta vertical (ou quase vertical) que separa p_{k+1} de K_k. Como consequência disso, temos sempre que p_k é um vértice de K_k. Além disso, p_{k+1} *vê* p_k, pois entre eles não há nenhum outro ponto de S.

Sendo assim, para identificar as arestas de K_k que devemos remover, podemos começar a percorrer a fronteira de K_k a partir de p_k. Fazemos isso partindo de p_k nas duas direções removendo arestas uv enquanto v esteja à esquerda ou sobre up,

(INCOMPLETO)

Exercícios.

1. Prove que a interseção de qualquer família de convexos é convexa.
2. Prove que p é ponto extremo de K se e somente se $p \in xy$ com $x, y \in K$ implica $p = x$ ou $p = y$.
3. Prove que $\text{conv}(S)$ é a união de todos os triângulos com vértices em S. Conclua que um ponto não é extremo se e somente se é combinação convexa não trivial de três pontos de S.
4. Prove que a região $y \geq x^2$ é uma região convexa, cuja fronteira é a parábola $y = x^2$.
5. Prove que o ponto mais baixo de um conjunto finito S é um ponto extremo de $\text{conv}(S)$.
6. Descreva um algoritmo que monta a fronteira de $\text{conv}(S)$ em tempo $O(n)$ a partir das suas arestas orientadas. Conclua que encontrar as arestas de $\text{conv}(S)$ requer tempo $\Omega(n \log n)$.

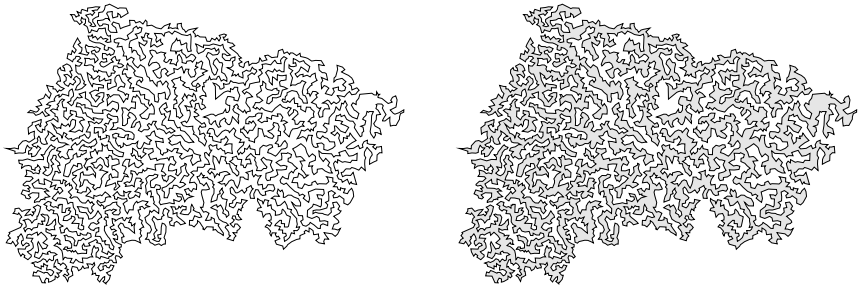
4. Localização de pontos

Vamos considerar agora o seguinte problema:

> *Dado um polígono simples* P *no plano e um ponto* p *do plano, localizar* p *em relação a* P*, isto é, decidir se* p *está no interior de* P*, no exterior de* P*, ou na fronteira de* P*.*

(Um polígono simples é uma linha poligonal fechada cujas arestas não se cruzam, a não ser nos vértices.)

Como mostra a figura abaixo, esse problema pode ser bem complicado. Note como o interior do polígono abaixo só fica evidente na figura da direita.



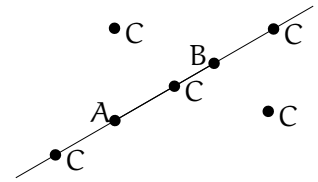
Apesar de parecer intuitivo, não é nada óbvio que um polígono simples realmente *tenha* um interior. Isso é na verdade um caso especial do teorema de Jordan:

> *Toda curva contínua fechada que não se cruza divide o plano em três regiões: a curva, o interior e o exterior.*

Portanto, o problema de localização de pontos é bem posto. Resta saber se ele tem uma solução algorítmica. Como veremos, a resposta é "sim" e a prova do teorema de Jordan nos dá uma solução algorítmica. Porém, antes de vermos a solução geral, vamos considerar alguns casos especiais do problema de localização de pontos.

Retas. O primeiro caso que vamos considerar é a localização de um ponto C em relação à reta definida por dois pontos A e B. Como no caso de polígonos, uma reta L no plano divide o plano em três regiões: a própria reta L e dois semi-planos. No caso do

polígonos, há uma diferença fundamental entre o interior e o exterior: o interior é limitado e o exterior é ilimitado. No caso de retas, ambos os semi-planos são ilimitados. Para distinguir os dois lados de L, orientamos L de A para B. Feito isso, podemos nos referir ao semi-plano da esquerda e ao semi-plano da direita. Localizar um ponto C em relação a L é portanto determinar se C está no semi-plano da esquerda, no semi-plano da direita, ou sobre L. Nesse último caso, podemos ainda distinguir cinco posições para C: antes de A, em A, entre A e B, em B, ou depois de B.



Esse é o principal predicado geométrico usado até agora. Vimos que ele foi essencial na solução do problema de fecho convexo. Vejamos agora os detalhes da sua implementação.

Quando C está à esquerda de AB, o vetor AC é obtido por uma rotação *positiva* do vetor AB em relação a A. Isso quer dizer que a componente de AC em relação ao complemento ortogonal positivo $AB^\perp$ de AB é positiva. O vetor $AB^\perp$ é o vetor obtido de AB por rotação de $+90°$ em relação a A, onde o sentido positivo de rotação é o sentido trigonométrico, contrário ao movimento do relógio. Portanto, $AB^\perp = (-(y_B - y_A), x_B - x_A)$. Assim, C está à esquerda de AB quando $\langle AC, AB^\perp \rangle > 0$.

Agora

$$\begin{aligned}
\langle AC, AB^\perp \rangle &= \langle (x_C - x_A, y_C - y_A), (-(y_B - y_A), x_B - x_A) \rangle \\
&= -(x_C - x_A)(y_B - y_A) + (y_C - y_A)(x_B - x_A) \\
&= \begin{vmatrix} x_B - x_A & x_C - x_A \\ y_B - y_A & y_C - y_A \end{vmatrix} \\
&= \begin{vmatrix} 1 & 0 & 0 \\ x_A & x_B - x_A & x_C - x_A \\ y_A & y_B - y_A & y_C - y_A \end{vmatrix} = \begin{vmatrix} 1 & 1 & 1 \\ x_A & x_B & x_C \\ y_A & y_B & y_C \end{vmatrix}.
\end{aligned}$$

Note que o determinante 2×2 acima é o que entendemos pelo produto vetorial $AB \times AC$ em $\mathbf{R}^2$. Assim, C está à esquerda de AB quando $AB \times AC > 0$; C está à direita de AB quando $AB \times AC > 0$; e C está sobre a reta AB quando $AB \times AC = 0$.

No caso degenerado $AB \times AC = 0$, o ponto C está sobre a reta AB e pode estar em cinco posições em relação ao segmento AB:

antes de A, em A, entre A e B, em B, ou depois de B. Se o segmento AB não for vertical, podemos usar as coordenadas x dos pontos A, B e C para localizar C relação a AB:

$$\begin{aligned} x_C < x_A &\Rightarrow \text{C está antes de A} \\ x_C = x_A &\Rightarrow \text{C está em A} \\ x_A < x_C < x_B &\Rightarrow \text{C está entre A e B} \\ x_C = x_B &\Rightarrow \text{C está em B} \\ x_B < x_C &\Rightarrow \text{C está depois de B} \end{aligned}$$

Se AB for vertical, então devemos testar as coordenadas y de modo análogo. Na prática, é melhor usar a coordenada correspondente à maior projeção de AB sobre os eixos coordenados.

Note que, mesmo para esse problema simples de localização de um ponto em relação a um segmento, temos muitos casos a tratar, principalmente devido às configurações degeneradas. Por outro lado, essa localização é uma primitiva importante e vale a pena o esforço de implementá-la de forma robusta e eficiente.

Como mencionamos na introdução, as configurações degeneradas ocorrem mesmo se não há erros de arredondamento. Por outro lado, mesmo com pontos de coordenadas inteiras, há problemas aritméticos na prática. De fato, para avaliar o determinante $AB \times AC$, precisamos fazer multiplicações. Isso pode levar a *overflow*, que para aritmética inteira geralmente é *silencioso*, isto é, o programa não é avisado da sua ocorrência.

Considere uma máquina típica que usa inteiros de 32 bits. Então ela pode representar todos os inteiros entre -2^{31} e $2^{31}-1$. Esses são números razoavelmente grandes: $2^{31} = 2147483648 \approx 2 \cdot 10^9$. Entretanto, se não queremos *overflow* no cálculo do determinante, ficamos limitados a 15 bits. De fato, considere o seguinte exemplo: $A = (0,0)$, $B = (b,b)$, $C = (-b,b)$, com $b > 0$. Então certamente C está à esquerda de AB pois $AB \times AC = 2b^2 > 0$. Para evitar *overflow*, devemos ter $2b^2 < 2^{31}$, isto é, $b < 2^{15} = 32768$, o que restringe bastante os números possíveis. O que acontece quando forçamos *overflow*? Para $b = 2^{15}$, temos $AB \times AC = 2^{31}$, que numa máquina de 32 bits é igual a -1. Concluímos que C está à direita de AB ! Felizmente, é possível calcular o *sinal* de $AB \times AC$ sem precisar calcular o seu valor, evitando portanto *overflow*.

Triângulos. O polígono mais simples é o triângulo. Seja então um ponto P que queremos localizar em relação a um triângulo ABC. Vamos assumir que o triângulo está orientado positivamente (isto é, $AB \times AC > 0$). Então o triângulo é a interseção dos três semiplanos positivos definidos pelas suas arestas. Em outras palavras, para saber se um ponto P está no interior de um triângulo ABC, basta localizar P em relação a cada uma das suas arestas: P está no interior do triângulo ABC se e somente se P está à esquerda de AB, BC, e CA. (Note a ordem das arestas!)

Uma outra solução para esse mesmo problema é usar as coordenadas baricêntricas. Essa técnica também é útil em outros contextos, e portanto vamos apresentá-la em detalhes.

As coordenadas baricêntricas vêm do seguinte resultado:

Sejam p_1, p_2 e p_3 pontos não colineares do plano $\mathbf{R}^2$. Então todo ponto p do plano pode ser escrito de modo único como combinação afim de p_1, p_2 e p_3, isto é, na forma

$$p = \lambda_1 p_1 + \lambda_2 p_2 + \lambda_3 p_3,$$

onde $\lambda_1, \lambda_2, \lambda_3$ são números reais satisfazendo $\lambda_1 + \lambda_2 + \lambda_3 = 1$.

A prova desse fato é bem simples. Basta observar que os ternos $(\lambda_1, \lambda_2, \lambda_3)$ satisfazendo as condições dadas são as soluções do seguinte sistema linear de três equações a três incógnitas:

$$\begin{array}{rcrcrcl} \lambda_1 x_1 & + & \lambda_2 x_2 & + & \lambda_3 x_3 & = & x \\ \lambda_1 y_1 & + & \lambda_2 y_2 & + & \lambda_3 y_3 & = & y \\ \lambda_1 & + & \lambda_2 & + & \lambda_3 & = & 1, \end{array}$$

onde $p = (x,y)$ e $p_i = (x_i, y_i)$, para $i = 1,2,3$.

O determinante desse sistema é

$$\begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{vmatrix} = p_1p_2 \times p_1p_3 \neq 0,$$

já que os pontos p_1, p_2, p_3 não são colineares. Portanto, o sistema dado tem solução única $(\lambda_1, \lambda_2, \lambda_3)$ para cada $p \in \mathbf{R}^2$. □

Os coeficientes λ_1, λ_2 e λ_3 dados acima são denominados as *coordenadas baricêntricas* de p em relação a p_1, p_2 e p_3. Esse nome

vem da seguinte observação: se massas iguais a λ_1, λ_2 e λ_3 são colocadas em p_1, p_2 e p_3, então o baricentro dessa configuração é o ponto $p = \lambda_1 p_1 + \lambda_2 p_2 + \lambda_3 p_3$.

Os valores de λ_1, λ_2 e λ_3 podem ser facilmente obtidos do sistema anterior utilizando a regra de Cramer. Temos, por exemplo,

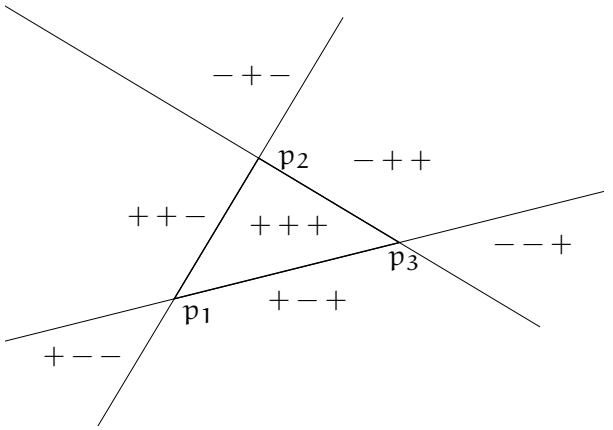
$$\lambda_1 = \frac{\begin{vmatrix} x & x_2 & x_3 \\ y & y_2 & y_3 \\ 1 & 1 & 1 \end{vmatrix}}{\begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{vmatrix}} = \frac{S(p, p_2, p_3)}{S(p_1, p_2, p_3)},$$

onde $S(p, p_2, p_3)$ e $S(p_1, p_2, p_3)$ são as áreas orientadas dos triângulos pp_2p_3 e $p_1p_2p_3$ respectivamente. Analogamente, temos

$$\lambda_2 = \frac{S(p_1, p, p_3)}{S(p_1, p_2, p_3)} \qquad e \qquad \lambda_3 = \frac{S(p_1, p_2, p)}{S(p_1, p_2, p_3)}.$$

As expressões acima permitem associar o sinal das coordenadas baricêntricas às regiões do plano determinadas pelas retas que contêm os lados do triângulo. Por exemplo, $\lambda_1 > 0$ se e só se o triângulo pp_2p_3 tem a mesma orientação de $p_1p_2p_3$, o que ocorre quando p está no mesmo semi-plano de p_1 em relação à reta que contém p_2p_3. Em particular, o triângulo corresponde à região $\lambda_1 \geq 0,\ \lambda_2 \geq 0,\ \lambda_3 \geq 0$, como era de se esperar, pois nesse caso p é uma combinação convexa de p_1, p_2 e p_3.

A figura abaixo mostra as diversas regiões do plano e os sinais correspondentes das coordenadas baricêntricas. Observe que a localização de um ponto em relação ao triângulo é imediata a partir dos sinais das suas coordenadas baricêntricas. Como essas coordenadas são facilmente obtidas a partir das primitivas que calculam áreas orientadas, o uso de coordenadas baricêntricas fornece uma forma compacta de efetuar a localização de pontos em relação a triângulos.

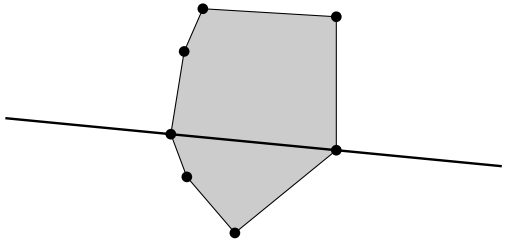


Polígonos convexos. Generalizando o que dissemos sobre triângulos, um polígono convexo é a interseção dos semiplanos positivos definidos pelas suas arestas. Mais precisamente, seja P um polígono convexo com vértices $p_1, \ldots, p_n$, orientado positivamente. Então um ponto p está dentro de P se e somente se p está à esquerda da aresta p_ip_{i+1} para todo $i = 1, \ldots, n$ (com a convenção de que $p_{n+1} = p_1$).

Portanto, temos um algoritmo simples que localiza um ponto em relação a um polígono convexo. Esse algoritmo leva tempo $O(n)$, pois temos n arestas para testar. Entretanto, podemos explorar a convexidade do polígono para melhorar bastante esse tempo.

De fato, considere a reta $L = p_1p_k$, onde $k = \lfloor n/2 \rfloor + 1$. Essa reta divide o polígono em duas partes iguais: $P^- = p_1p_2 \ldots p_k$ e $P^+ = p_1p_kp_{k+1} \ldots p_n$, cuja interseção é a diagonal p_1p_k. Se p estiver sobre L e dentro dessa diagonal, então p está em P. Se p estiver sobre L mas fora dessa diagonal, então p está fora de P. Se p não estiver sobre L, então podemos considerar as duas partes P^- e P^+ como disjuntas, pois p só pode estar em umas delas, se estiver dentro do polígono P. Cada uma dessas partes é um polígono convexo, orientado positivamente, com metade dos vértices de P. Se p estiver à esquerda de L (isto é, à esquerda de p_1p_k), então basta localizar p em relação ao polígono convexo P^+. Se p estiver

à direita de L, então basta localizar p em relação ao polígono convexo P^-. Repetimos recursivamente esse processo com P^+ ou P^-, até chegar a um triângulo, pois já sabemos localizar pontos nesse caso. Como em ambos os casos reduzimos à metade o tamanho do problema, após $O(\log n)$ passos teremos terminado.



(INCOMPLETO)

Exercícios.

1. Mostre que as coordenadas baricêntricas de um ponto podem ser interpretadas através da imagem da transformação afim $T\colon \mathbf{R}^2 \to \mathbf{R}^3$ tal que $T(p_1) = (1,0,0)$, $T(p_2) = (0,1,0)$, $T(p_3) = (0,0,1)$. Essa transformação é injetiva e leva o plano $\mathbf{R}^2$ (definido pela equação $z = 0$) no plano definido pela equação $x + y + z = 1$ no $\mathbf{R}^3$.
2. Mostre que não existe uma generalização natural de coordenadas baricêntricas para polígonos convexos com mais de três lados, pois não temos unicidade.

5. Interseção de segmentos

Vamos considerar agora o seguinte problema:

> *É dada um cena com n segmentos de reta no plano. Queremos saber se há cruzamentos e quais são eles.*

Temos portanto dois problemas:

DETECÇÃO
: Determinar se existe um par de segmentos que se cruza.

IDENTIFICAÇÃO
: Determinar todos os pares de segmentos que se cruzam.

Consideremos primeiro o caso $n = 2$, isto é, como decidir se dois segmentos no plano se cruzam. Saber resolver esse caso é essencial para resolver o caso geral.

Dois segmentos. Uma maneira intuitiva de decidir se dois segmentos AB e CD se cruzam é tentar encontrar o ponto onde eles se cruzam. Isso certamente resolve o problema mas é mais do que queremos fazer, pois o problema não exige encontrar o ponto de cruzamento. Apesar disso, vamos seguir nessa direção.

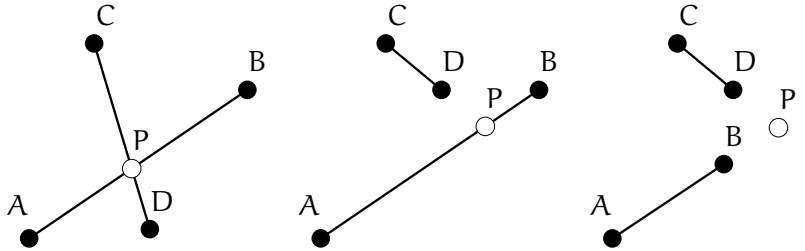
Uma estratégia óbvia para o ponto onde os segmentos AB e CD se cruzam é encontrar o ponto de cruzamento das *retas* AB e CD e depois verificar se esse ponto está dentro de ambos os segmentos. A reta AB pode ser parametrizada por $A + t(B - A)$ com $t \in \mathbf{R}$. A reta CD pode ser parametrizada por $C + s(D - C)$ com $s \in \mathbf{R}$. Assim, existe um ponto P comum às duas retas quando existem $t, s \in \mathbf{R}$ tais que $A + t(B - A) = P = C + s(D - C)$. Expandindo essa equação em coordenadas, o cálculo de P se resume à solução de um sistema linear 2×2:

$$\begin{aligned} x_A + t(x_B - x_A) &= x_C + s(x_D - x_C) \\ y_A + t(y_B - y_A) &= y_C + s(y_D - y_C), \end{aligned}$$

que em forma matricial fica:

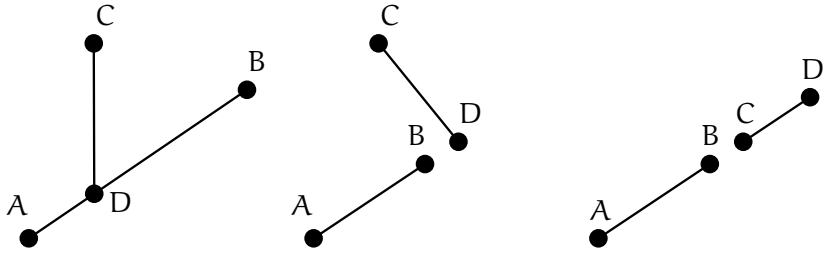
$$\begin{pmatrix} x_B - x_A & x_C - x_D \\ y_B - y_A & y_C - y_D \end{pmatrix} \begin{pmatrix} t \\ s \end{pmatrix} = \begin{pmatrix} x_C - x_A \\ y_C - y_A \end{pmatrix}.$$

Esse sistema tem solução se e somente se os segmentos não são paralelos, e nesse caso tem solução única $t, s \in \mathbf{R}$. Como queremos saber se o ponto de cruzamento está dentro dos segmentos, concluímos que os segmentos se cruzam se e somente se $t, s \in [0, 1]$.



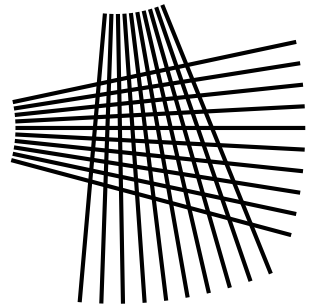
Se precisamos saber o ponto de cruzamento, então não há nada muito diferente a fazer. Porém, se só queremos saber se os segmentos se cruzam, sem precisar saber o ponto de cruzamento, então é possível fazer algo mais simples: AB e CD se cruzam exatamente quando C e D estão de lados opostos de AB e A e B estão de lados opostos de CD. Mais uma vez, a localização de pontos em relação a segmentos é a primitiva geométrica fundamental. Note que ambas as condições são necessárias, como mostra o exemplo do meio na figura acima.

O critério de lados opostos é surpreendentemente robusto nos casos degenerados: basta interpretar corretamente "lados opostos". Por exemplo, se C está fora de AB e D está dentro de AB, então eles estão de lados opostos. Por outro lado, se C está fora de AB e D está na reta AB mas fora de AB, então eles não estão de lados opostos. Se C e D estão ambos sobre a reta AB, então a classificação deles em relação ao segmento AB é consistente com a noção de "lados opostos".



Em resumo, a interseção de dois segmentos de reta no plano pode ser decidida completamente, de uma maneira razoavelmente simples, ainda que haja vários casos a serem tratados.

O caso geral. Voltemos ao caso geral, uma cena com n segmentos no plano. Ambos os problemas, DETECÇÃO e IDENTIFICAÇÃO, admitem uma solução trivial, de força bruta: testar todos os pares. Como temos $\binom{n}{2}$ pares de segmentos, essa solução leva tempo $O(n^2)$. Para IDENTIFICAÇÃO, não podemos melhorar esse limite superior, porque existem cenas que têm $\Omega(n^2)$ cruzamentos. Concluímos então que o algoritmo de força bruta é ótimo para IDENTIFICAÇÃO!



Por que podemos esperar fazer melhor do que isso? Em primeiro lugar, porque o limite inferior $\Omega(n^2)$ para IDENTIFICAÇÃO não se aplica para DETECÇÃO. Para DETECÇÃO temos por enquanto somente o limite inferior trivial, $\Omega(n)$. Em segundo lugar, o número de cruzamentos I pode ser (e na prática é) muito menor do que $O(n^2)$. É natural portanto considerar a complexidade de IDENTIFICAÇÃO também em função de I, que é o tamanho da resposta. Assim, temos um limite inferior trivial $\Omega(n + I)$ para IDENTIFICAÇÃO.

Para estabelecer um limite inferior não trivial, precisamos de um problema para servir de base, se quisermos usar a técnica de redução. Infelizmente, o único problema que temos para isso é ORDENAÇÃO, mas não parece ser possível reduzir esse problema a DETECÇÃO ou IDENTIFICAÇÃO. Um problema tão básico quanto ORDENAÇÃO e tem realmente ligação com esses problemas de interseção de segmentos é o problema de unicidade:

> *Dados n números reais $x_1, x_2, \ldots, x_n \in \mathbf{R}$, decidir se há duplicatas entre eles, isto é, se existem i e j com $i \neq j$ tais que $x_i = x_j$.*

Esse problema pode ser resolvido simplesmente ordenando os números e depois verificando se há dois números consecutivos

que são iguais. Em outras palavras, UNICIDADE se reduz a ORDENAÇÃO: se sabemos ordernar, sabemos decidir unicidade. Essa redução implica então que UNICIDADE é $O(n \log n)$, pois ORDENAÇÃO pode ser resolvido nesse tempo. A redução *não* implica que UNICIDADE é $\Omega(n \log n)$, embora ORDENAÇÃO seja $\Omega(n \log n)$. Para isso, seria preciso fazer a redução na direção contrária, Entretanto, não se conhece uma redução de ORDENAÇÃO para UNICIDADE, nem isso parece ser possível: afinal, como gerar uma saída de tamanho n a partir de uma resposta "sim/não"? Por outro lado, prova-se diretamente que UNICIDADE é $\Omega(n \log n)$, mas a prova não é elementar. Esse é um resultado importante e que merece destaque:

> *No modelo de árvores de decisões algébricas, qualquer algoritmo que decida se há duplicatas numa sequência de n números leva tempo $\Omega(n \log n)$.*

Usando esse resultado, podemos obter um limite inferior não trivial para DETECÇÃO, reduzindo UNICIDADE a DETECÇÃO. De fato, para testar a unicidade de n números reais $x_1, \ldots, x_n$, construa n segmentos de reta verticais $s_1, \ldots, s_n$, onde $s_i = \{x_i\} \times [0, i]$. Então, $x_i = x_j$ se e somente se $s_i \cap s_j \neq \varnothing$. Portanto, DETECÇÃO tem complexidade $\Omega(n \log n)$.

Se sabemos resolver IDENTIFICAÇÃO, então também sabemos resolver DETECÇÃO: basta ver se a lista de cruzamentos é vazia. DETECÇÃO se reduz a IDENTIFICAÇÃO e podemos transferir o limite inferior $\Omega(n \log n)$ de DETECÇÃO para IDENTIFICAÇÃO. Entretanto, como vimos no exemplo acima, esse limite inferior não é ser o melhor: no pior caso, IDENTIFICAÇÃO é $\Omega(n^2)$. Por outro lado, incluindo o número de cruzamentos I na complexidade, obtemos o limite inferior não trivial $O(n \log n + I)$.

Vamos descrever agora um algoritmo que resolve DETECÇÃO em tempo ótimo $\Omega(n \log n)$. Esse algoritmo pode ser estendido para resolver IDENTIFICAÇÃO em tempo $O((n + I) \log n)$, que não é ótimo mas é quase. Existem algoritmos ótimos para IDENTIFICAÇÃO, mas são muito complicados.

(INCOMPLETO)

6. Par mais próximo

Vamos considerar agora o seguinte problema:

> *Dentre n pontos no plano, encontrar o par mais próximo, isto é, aquele que determina a menor distância.*

Assim como no problema de interseção de segmentos, esse problema admite uma solução trivial, de força bruta: testar todos os pares. Como temos $\binom{n}{2}$ pares de pontos, essa solução leva tempo $O(n^2)$. Podemos esperar fazer melhor do que isso?

É fácil estabelecer um limite inferior não trivial para PAR MAIS PRÓXIMO reduzindo UNICIDADE a esse problema. De fato, para testar a unicidade de n números reais $x_1, \ldots, x_n$, defina n pontos no plano $p_1, \ldots, p_n$, onde $p_i = (x_i, 0)$. Então, há duplicatas em $x_1, \ldots, x_n$ se e somente se o par mais próximo dentre $p_1, \ldots, p_n$ está a distância 0. Podemos então transferir o limite inferior não trivial $\Omega(n \log n)$ de UNICIDADE para PAR MAIS PRÓXIMO.

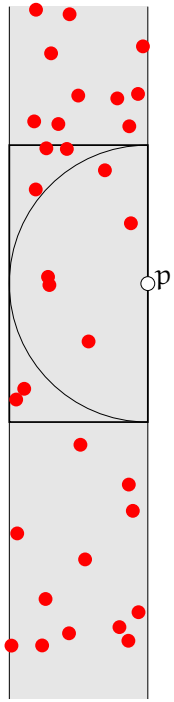
Por que podemos esperar resolver PAR MAIS PRÓXIMO sem testar todos os pares? Considere uma abordagem incremental: processamos os pontos $p_1, \ldots, p_n$ um de cada vez, na ordem em que eles são dados. A cada passo, identificamos o par mais próximo dentre os pontos já vistos. No final, claro, teremos indentificado o par mais próximo dentre todos os pontos.

Vejamos essa abordagem em detalhe. O primeiro passo é calcular a distância em p_1 e p_2. No passo k, teremos processado $p_1, \ldots, p_{k-1}$ e encontrado a menor distância δ entre esses pontos. Temos agora que processar p_k. Então duas coisas podem acontecer: ou p_k está a uma distância menor do que δ de um dos outros pontos ou não. No primeiro caso, o par mais próximo em $p_1, \ldots, p_k$ é $p_i p_k$, onde p_i é o ponto mais próximo de p_k com $i < k$. No segundo caso, o par mais próximo em $p_1, \ldots, p_k$ é o par mais próximo em $p_1, \ldots, p_{k-1}$, que já conhecemos. Em outras palavras, o par mais próximo muda ao processarmos p_k se e somente se existem pontos de $\{p_1, \ldots, p_{k-1}\}$ dentro do círculo de raio δ centrado em p_k. Nesse caso, o novo par mais próximo é determinado pelo ponto mais próximo de p_k dentro desse círculo.

Assim, temos uma novamente noção de zona de influência — o círculo de raio δ centrado em p_k — que varia durante o pro-

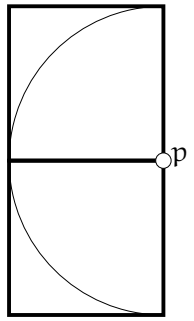
cesso. Repetindo: se já conhecemos a distância δ entre dois pontos, então não precisamos considerar pares de pontos que estejam a uma distância maior do que δ. O problema é como identificar os pares que precisam ser considerados sem ter que olhar para todos os pares. A solução é fazer uma exploração sistemática dos pontos, guiada por uma estimativa dinâmica da menor distância. Como vimos no problema de interseção de segmentos, uma técnica para uma tal exploração sistemática é a técnica de varredura.

Assim, vamos expor agora uma abordagem incremental por varredura para PAR MAIS PRÓXIMO. A abordagem é incremental como descrito acima: os pontos são processados um de cada vez, e obtemos a solução parcial, correspondente aos pontos já processados. Nesse caso, a solução parcial é o par mais próximo a cada passo. Como vamos usar varredura, os pontos são processados da esquerda para a direita, em ordem lexicográfica. Essa é a fila de eventos da varredura.



Mas qual é o estado da varredura? Certamente, o par mais próximo dentre os pontos já visitados tem que fazer parte do estado. Como discutimos acima, esse par só muda durante a varredura quando um novo ponto é processado, isto é, a cada evento. Se o par muda ao processarmos um ponto p é porque existe um ponto q já processado que está a um distância menor do que δ de p, onde δ é a menor distância entre os pontos antes de p. Como os pontos estão ordenados por x, os únicos pontos que têm chance de formar um novo par mais próximo são aqueles pontos q que aparecem na fila de eventos a menos de δ de p, isto é, tais que $x_p - x_q < \delta$. De fato, se a distância entre p e q é menor do que δ, então a distância entre as suas projeções no eixo x também é menor do que δ. Essa observação nos leva a manter durante a varredura uma *faixa* vertical de largura δ, à esquerda da reta de varredura. Somente pontos dentro dessa faixa vão poder formar um novo par mais próximo. É

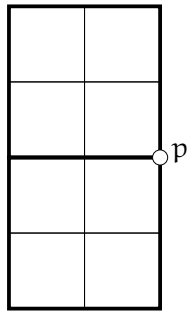
nesse sentido que vamos controlar zonas de influência dinamicamente.



Uma observação semelhante nos diz que os únicos pontos q dessa faixa que têm chance de formar um novo par mais próximo com p são aquelas cuja distância em y também é menor do que δ. Ou seja, todos os candidatos a formar um novo par mais próximo com p estão num retângulo de largura δ e altura 2δ. Para poder encontrar eficientemente esses pontos, vamos então manter os pontos da faixa vertical ordenados em y numa lista L. Desse modo, para encontrar os pontos candidatos q, basta considerar os *vizinhos* de p em L tais que $|y_p - y_q| < \delta$.

Resumindo essa discussão, a varredura é feita com os seguintes passos:

1. Criamos a fila de eventos ordenando os pontos lexicograficamente (em x e depois em y).
2. A varredura começa em p_2 com $\delta = d(p_1, p_2)$. A faixa vertical começa em $[x_1, x_2]$. A lista L começa com p_1 e p_2, ordenados por y.
3. No passo k, trazemos a reta de varredura para x_k e a faixa vertical de $[x_i, x_k]$ para $[x_j, x_k]$, eliminando de L os pontos p_u com $i \leq u < j$ para os quais $p_k - p_u \geq \delta$.
4. Incluímos p_k em L e verificamos dentre os vizinhos q de p_k em L se há algum que $d(p_k, q) < \delta$, atualizando o par mais próximo e δ se encontrarmos um tal ponto q.



A observação crucial para o desempenho desse algoritmo é que não temos que considerar muitos candidatos no passo 4. Na verdade, existem no máximo 8 candidatos! De fato, todos os candidatos estão num retângulo de largura δ e altura 2δ. Esse retângulo pode ser decomposto em 8 quadrados de lado $\delta/2$. Como todos os pontos dentro do retângulo têm que distar no mínimo δ entre si, temos no máximo *um* ponto em cada quadrado. Como são 8 quadrados, temos no máximo 8 candidatos.

Feita essa observação, podemos calcular o tempo que o algoritmo leva. Para a criação da fila de eventos no passo 1, precisamos ordenar os pontos. Isso leva tempo $O(n \log n)$. Podemos manter a lista L em tempo $O(\log n)$ por operação usando uma árvore balanceada. O número total máximo de remoções feitas no passo 3 é n; logo o tempo total gasto nesse passo é $O(n \log n)$. No passo 4, a inclusão de p_k em L e a identificação de cada vizinho candidato levam tempo $O(\log n)$. Como temos no máximo 8 candidatos, o tempo total gasto nesse passo é $O(n \log n)$. Assim, o algoritmo leva tempo total $O(n \log n)$ e portanto é ótimo!

Mais uma vez, note como a combinação de argumentos geométricos com estruturas de dados adequadas permitiu uma solução ótima do problema, fazendo uma exploração sistemática das zonas de influência de maneira dinâmica.

7. Circulo mínimo

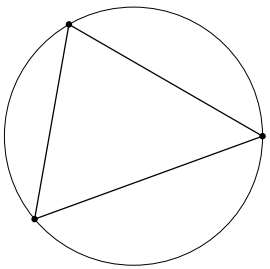
Vamos considerar agora o seguinte problema:

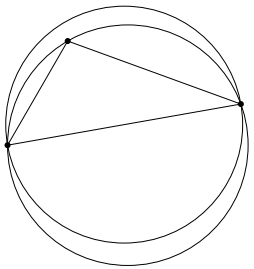
> *Dados n pontos no plano, encontrar o menor círculo que contém todos eles.*

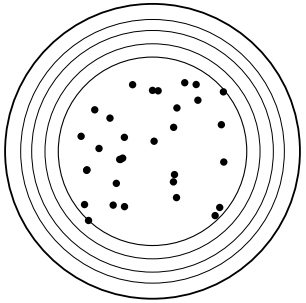
Esse é um problema *construtivo*: temos que encontrar o centro e o raio de menor círculo que contém todos os pontos dados. Só raramente esse centro vai ser um dos pontos originais.

Antes de estudarmos soluções algorítmicas para esse problema, temos que estabelecer que o problema realmente tem uma solução. Para outros problemas que estudamos, isso foi fácil porque só tínhamos um número finito de candidatos possíveis a solução. Mas para o problema de círculo mínimo, temos um número infinito de candidatos possíveis! Vamos então estudar a geometria do problema para descobrir se é possível reduzir a lista de candidatos a um número finito.

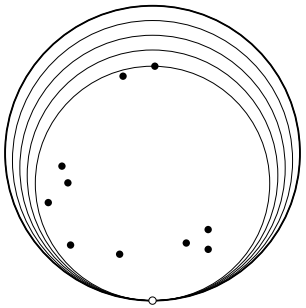
Começamos com os casos simples: $n \leq 3$. Para $n = 1$, o círculo mínimo tem centro p_1 e raio zero. Para $n = 2$, o círculo mínimo tem centro no ponto médio $(p_1 + p_2)/2$ e raio $d(p_1, p_2)/2$, isto é, o círculo com diâmetro p_1p_2. O caso $n = 3$ já não é tão simples. Se os três pontos forem colineares, digamos com p_2 entre p_1 e p_3, então o círculo mínimo têm diâmetro p_1p_3. Se os três pontos não forem colineares, um círculo que contém todos os pontos é o circuncírculo do triângulo $p_1p_2p_3$: ele tem centro no baricentro $(p_1 + p_2 + p_3)/3$ e raio igual à distância do baricentro a qualquer um dos pontos. Entretanto, o circuncírculo não é sempre o menor círculo que contém os três pontos: se o triângulo $p_1p_2p_3$ tem um ângulo obtuso, digamos em p_2, o círculo mínimo têm diâmetro p_1p_3, pois p_2 está dentro desse círculo. Passemos ao caso geral.





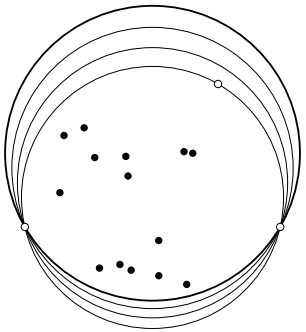


Seja S um conjunto finito de n pontos no plano. Certamente existe pelo menos um círculo que contém S: basta tomar qualquer ponto do plano como centro e um raio suficientemente grande. Se esse círculo não passa por nenhum ponto de S, então ele não pode ser o círculo mínimo, pois podemos reduzí-lo mantendo o centro O mas tomando como raio a distância de O ao ponto de S mais longe de O. Em outras palavras, se um círculo contém S mas não tem pelo menos um *ponto de contato* com S, então ele não pode ser o círculo mínimo.



Na verdade, se um círculo contém S mas não tem pelo menos *dois* pontos de contato, então ele não pode ser o círculo mínimo. De fato, suponha que temos um círculo contendo S com somente um ponto de contato A. Então podemos reduzir o círculo, mantendo-o em contato com A, movendo o centro O do círculo na direção radial de A até encontrar um outro ponto de contato B.

Concluímos então que existe um círculo contendo S e com pelo menos dois pontos de contato A e B. Se AB é um diâmetro do círculo, então o círculo é o círculo mínimo que contém S, pois

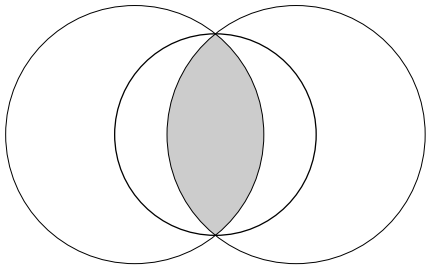


nenhum círculo menor pode conter A e B. Se AB não é um diâmetro do círculo e não há outros pontos de contato, então podemos reduzir o círculo, mantendo-o em contato com A e B, movendo o centro na mediatriz de AB em direção a AB até encontrar um terceiro ponto de contato C. Assim, existe um círculo contendo S e com pelo menos três pontos de contato: A, B e C.

Resumindo: para encontrar o menor círculo que contém S, basta considerar todos os círculos definidos por dois ou três pontos de S e que contêm todos os pontos de S. De fato, se encontramos um círculo definido por dois pontos de S e contendo S, então ele tem que ser o círculo mínimo. Se nenhum dos círculos definidos por dois pontos de S contêm todos os pontos de S, então todos os círculos que contêm S são definidos por três pontos de S. Considere o menor desses círculos. Então os três pontos de S que definem o círculo têm que determinar um triângulo cujos ângulos são todos agudos; caso contrário, seria possível reduzir o círculo, como vimos na discussão do caso $n = 3$. Esse resultado merece um destaque:

> *O menor círculo contendo um conjunto finito S de pontos do plano é definido por dois pontos de S diametralmente opostos ou por três pontos de S formando um triângulo cujos ângulos são todos agudos.*

Desse resultado vem então um algoritmo de força bruta: Considere todos os pares e triplas de pontos de S. Escolha o menor círculo definido por dois ou três pontos de S e que contém todos os pontos de S. Como temos $\binom{n}{2}$ pares de pontos e $\binom{n}{3}$ triplas de pontos de S, e devemos testar todos os pontos de S para cada círculo candidato, esse algoritmo leva tempo $n(\binom{n}{2} + \binom{n}{3}) = O(n^4)$. Esse algoritmo não é eficiente, mas serve para mostrar a existência do círculo mínimo. Aproveitamos para mostrar a unicidade. De fato, se tivéssemos dois círculos mínimos diferentes, então eles teriam centros diferentes mas mesmo raio. Mas então S estaria na interseção desses círculos e seria possível encontrar um círculo menor contendo S:

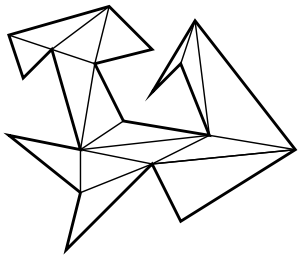


8. Triangulação de polígonos

Vamos considerar agora o seguinte problema:

> *Dado um polígono simples com n lados, decompor o seu interior em triângulos por meio de diagonais.*

Uma *diagonal* de um polígono é um segmento ligando dois vértices não consecutivos e que está totalmente contido no interior do polígono.



Esse problema é fácil para polígonos convexos, mas não é óbvio que ele tenha solução em geral. Muitos livros e artigos, incluindo livros de geometria clássica, usam a existência de triangulações para polígonos sem demonstrá-la ou mesmo mencioná-la. Vale então dar um destaque a esse fato:

> *Todo polígono simples admite uma triangulação. Além disso, se o polígono tem n lados, então toda triangulação tem $n-2$ triângulos e usa $n-3$ diagonais.*

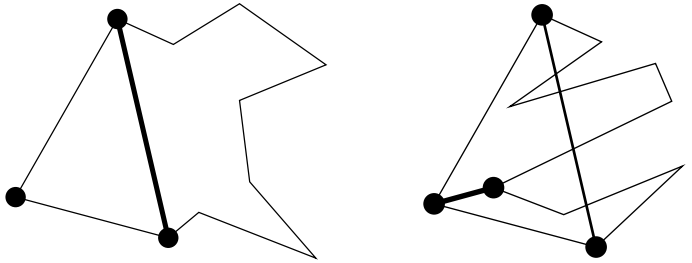
A prova desse fato é por indução em n. O caso base é $n = 3$: o polígono então é um triângulo e não tem nenhuma diagonal. Portanto a afirmativa é trivialmente verdadeira nesse caso. O passo de indução depende da seguinte observação crucial:

> *Todo polígono com mais de três lados tem uma diagonal.*

Assumindo por enquanto esse fato, podemos completar a indução: Se P um polígono de n lados com $n > 3$. Então podemos dividir P por meio de uma diagonal, obtendo dois polígonos P_1 e P_2, cada um com menos de n lados. Pela hipótese de indução, P_1 e P_2 podem ser triangulados por meio de diagonais. As diagonais usadas nessas triangulações também são diagonais de P.

Como $P = P_1 \cup P_2$, a combinação das triangulações de P_1 e P_2 é uma triangulação de P. Além disso, se P_i tem n_i lados, então temos também pela hipótese de indução que a triangulação de P_i tem $n_i - 2$ triângulos e usa $n_i - 3$ diagonais. Como $n_1 + n_2 = n + 2$, concluímos que o número de triângulos na triangulação de P é $n_1 - 2 + n_2 - 2 = n_1 + n_2 - 4 = n + 2 - 4 = n - 2$ e o número de diagonais é $n_1 - 3 + n_2 - 3 + 1 = n_1 + n_2 - 5 = n + 2 - 5 = n - 3$. □

Resta então mostrar a existência de pelo menos uma diagonal. Para isso seja v um vértice convexo de P, por exemplo o vértice mais à esquerda. Sejam u e w os vizinhos de v em P. Considere o triângulo uvw. Se não há outros vértices de P dentro desse triângulo, então uw é uma diagonal de P. Caso contrário, seja x o vértice de P dentro do triângulo uvw que está mais longe do segmento uw. Então vx é uma diagonal de P. □



A existência de uma diagonal pode parecer óbvia. Entretanto, ela não vale em dimensão 3: existem poliedros com mais de quatro vértices que não têm nenhuma diagonal! Além disso, decidir se um dado poliedro tem ou não uma diagonal é um problema muito difícil (NP-difícil).

Uma consequência da existência de triangulações é o seguinte fato muito conhecido, mas geralmente enunciado somente para polígonos convexos (dando a entender que ele não vale em geral):

> *A soma dos ângulos internos de um polígono simples de n lados é $(n-2)\pi$.*

De fato, considere uma triangulação do polígono. Então temos $n - 2$ triângulos. Como a soma dos ângulos internos de um triângulo é π, a soma dos ângulos internos do polígono é $(n-2)\pi$. □

Uma outra consequência da existência de triangulações é uma fórmula para área de um polígono, que é uma versão discreta do teorema de Green do cálculo:

Seja $P = p_1 p_2 \ldots p_n$ *um polígono simples no plano e* q *um ponto qualquer do plano. Então, a área de* P *é dada por*

$$A(P) = \sum_{i=1}^{n} A(q p_i p_{i+1}).$$

Em particular, tomando q *como a origem, temos*

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

De fato, dividindo o polígono em duas partes por meio de uma diagonal uv e usando indução, obtemos duas somas, uma com a parcela $A(quv)$ e outra com a parcela $A(qvu) = -A(quv)$. Essas parcelas portanto se cancelam, sobrando somente as parcelas correspondentes aos lados do polígono. □

A prova da existência de triangulações nos dá um algoritmo para encontrar uma triangulação:

1. Encontre uma diagonal.
2. Divida o polígono em dois polígonos menores cortando na diagonal.
3. Triangule recursivamente as duas partes.

Os passos 1 e 2 podem ser feitos em tempo $O(n)$. O passo 3 leva tempo $T(n_1)+T(n_2)$. O tempo total então é $T(n) = T(n_1)+T(n_2)+cn$. Se $n_1 \approx n_2$, então a solução é $T(n) = O(n \log n)$, como vimos quando estudamos ordenação por intercalação. Entretanto, é possível que tenhamos $n_1 = 3$ e $n_2 = n-1$: é o que acontece com polígonos convexos. Nesse caso, temos $T(n) = \Theta(n^2)$. Em outras palavras, os polígonos convexos são o pior caso para esse algoritmo, ainda que polígonos convexos possam ser facilmente triangulados em tempo linear.

Para termos $T(n) = O(n \log n)$ no pior caso é necessário saber encontrar eficientemente uma diagonal que divida o polígono em partes aproximadamente iguais. É possível encontrar em tempo linear uma diagonal tal que $n_i \geq n/3$, mas não sabemos se é possível fazer isso sem triangular o polígono primeiro.

De qualquer modo, mostramos até agora que todo polígono simples pode ser triangulado em tempo $O(n^2)$. Os polígonos convexos podem ser triangulados em tempo $O(n)$, mas isso não tem consequência direta para o caso geral pois é comum que problemas possam ser resolvidos rapidamente para polígonos convexos mas não para polígonos simples arbitrários. Fica então a questão: quão rápido podemos triangular um polígono simples?

Não temos nenhum limite inferior óbvio, além do limite trivial $\Omega(n)$. Na verdade, não existe nenhum limite inferior maior do que esse porque existem algoritmos que triangulam polígono simples arbitrários em tempo $O(n)$. Durante muitos anos, a busca de algoritmos lineares para esse problema foi o problema em aberto mais importante em geometria computacional. A solução foi dada em 1990 por Chazelle, mas o algoritmo e as estruturas de dados usados nessa solução são complicadíssimos e nada práticos. (Até onde se sabe, o algoritmos de Chazelle nunca foi implementado.) A busca por algoritmos práticos continua.

Galerias de arte. Como aplicação de triangulações de polígonos simples, consideremos o seguinte problema "prático": É dada uma galeria de arte cuja planta é um polígono simples. Queremos instalar câmeras de segurança no teto da galeria de modo a poder vigiar todas as obras de arte em exposição. Vamos supor que as obras de arte são quadros pendurados na parede e que as câmeras são instaladas em pontos fixos mas podem rodar 360°. Quantas câmeras são necessárias? Aonde instalar as câmeras?

Dado um polígono simples P, seja $G(P)$ o número mínimo de câmeras necessárias para vigiar P. Então claramente $G(P)$ depende de P. Por exemplo, se P é convexo, então $G(P) = 1$. Mas é possível ter $G(P) = 1$ para P não convexo: basta que exista um ponto de P que veja todos os outros. Quando isso acontece, dizemos que P é *estrelado*. (Certamente, todo polígono convexo é

estrelado, mas a recíproca não é verdadeira.) Os polígonos estrelados são exatamente aqueles que têm $G = 1$.

Vejamos um limite superior para $G(P)$. Como todo polígono simples pode ser triangulado com $n - 2$ triângulos, e triângulos são convexos, podemos vigiar qualquer polígono simples de n lados usando $n - 2$ câmeras: basta triangular o polígono e por uma câmera em cada triângulo. Portanto, $G(P) \leq n - 2$ para qualquer polígono simples de n lados. Mas será que existe algum polígono para o qual $n - 2$ câmeras são realmente necessárias?

O valor exato de $G(P)$ varia muito com P, mesmo entre os polígonos simples com o mesmo número de lados. Na verdade, calcular $G(P)$ é um problema muito difícil (NP-difícil). Sendo assim, vamos considerar o pior caso e ver G com função de n:

$$G(n) = \max_{|P|=n} G(P).$$

Então, como vimos, temos $1 \leq G(n) \leq n - 2$. Entretanto, vamos provar o seguinte resultado mais forte:

$$G(n) = \lfloor n/3 \rfloor.$$

Vamos provar isso em duas partes: $G(n) \geq \lfloor n/3 \rfloor$ e $G(n) \leq \lfloor n/3 \rfloor$.

Para provar a primeira parte, $G(n) \geq \lfloor n/3 \rfloor$, basta exibir um polígono de n lados que precise de pelo menos $\lfloor n/3 \rfloor$ câmeras. O polígono abaixo tem k picos e $n = 3k$ lados.

figura – polígono com k picos

Considere agora os pontos que vêem um dado pico. Escolhendo com cuidado o ângulo nos picos, concluímos que se um ponto vê um dado pico então ele não vê nenhum outro pico. Como temos k picos, precisamos de pelo menos $k = \lfloor n/3 \rfloor$ câmeras, pois cada pico tem que ser visto por pelo menos uma câmera. Portanto, $G(n) \geq G(P) \geq k = \lfloor n/3 \rfloor$. (Na verdade, $G(P) = k$, isto é, k câmeras são suficientes para vigiar esse polígono, mas isso não é essencial.)

figura – polígono com k câmeras

Mostrar a segunda parte, $G(n) \leq \lfloor n/3 \rfloor$, é bem mais interessante.

Seja P um polígono simples com n lados. Construa uma triangulação de P. Como vimos, essa triangulação tem $n-2$ triângulos e instalar uma câmera em cada triângulo. Se instalarmos as câmeras nos vértices de P, então é muito provável que uma câmera consiga vigiar mais de um triângulo. Vamos mostrar que é possível escolher $\lfloor n/3 \rfloor$ vértices e ainda conseguir vigiar todo o polígono.

A ferramenta que vamos usar é uma *coloração*. A observação crucial é que é possível colorir os vértices de P usando três cores de modo que dois vértices adjacentes na triangulação nunca tenha a mesma cor. Veremos como fazer isso logo, mas antes vejamos como isso resolve o nosso problema de $\lfloor n/3 \rfloor$ vértices aonde instalar câmeras de modo a vigiar todo o polígono.

Sejam A, B e C o número de vértices de cada uma das três cores. Então claramente temos $A + B + C = n$. Portanto não podemos ter $A > n/3$ e $B > n/3$ e $C > n/3$, pois se caso contrário teríamos $A + B + C > n$. Ou seja, pelo menos uma das três cores é usada em no máximo $\lfloor n/3 \rfloor$ vértices. Portanto, instalar câmeras nos vértices coloridos com a cor menos usada. Assim, cada triângulo da triangulação de P será completamente vigiado por um vértice dessa cor e assim P será completamente vigiado por $\lfloor n/3 \rfloor$ câmeras. □

Resta então mostrar como colorir a triangulação usando somente três cores. (O famoso teorema das quatro cores diz que todo mapa pode ser colorido com quatro cores. Estamos dizendo aqui que bastam três cores quando o mapa é uma triangulação de um polígono simples.) Mas isso é fácil usando indução: Ache aresta da triangulação que seja uma diagonal. Divida o polígono em duas partes. Colora as duas partes usando a hipótese de indução. Acerte as cores na colagem.

O único passo não trivial é a colagem. Seja uv a diagonal escolhida. Sejam P_1 e P_2 as duas partes de P que se colam em uv. Por indução, sabemos colorir as triangulações de P_1 e P_2 usando três cores. Se tivermos sorte e as cores usadas para uv em P_1 forem exatamente as mesmas usadas em P_2, então teremos colorido P

com três cores. Se as cores não forem as mesmas, basta fazer uma permutação das cores usadas em P_2 de modo que elas fiquem as mesmas usadas em P_1.

Um outro modo de fazer essa coloração é usar que toda triangulação de um polígono simples tem uma *orelha*, isto é, um triângulo correspondente a um folha no grafo dual. Corte essa orelha fora, obtendo uma triangulação de um polígono com $n-1$ lados. Por indução, colora esse polígono com três cores. Adicione o triângulo cortado, colorindo o vértice cortado com a cor não usada na aresta de colagem. Usando busca em profundidade no dual da triangulação, esse procedimento pode ser feito em tempo $O(n)$.

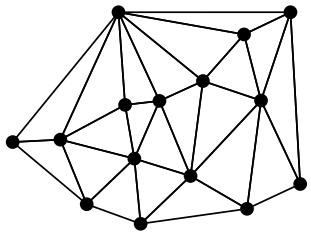
Exercícios.

1. Quantas triangulações admite um polígono convexo de n lados?
2. Existem polígonos que só admitem uma triangulação?
3. Na prova da existência de uma diagonal, mostre que não funciona escolher x como o vértice de P mais perto de v.
4. Prove que toda região poligonal mesmo com buracos pode ser triangulada por meio de diagonais. Mostre que a região tem h buracos, então toda triangulação tem $n-2h-2$ triângulos. E quantas diagonais?
5. Prove que a soma dos ângulos externos de um polígono simples é 2π.
6. Mostre como triangular um polígono convexo em tempo linear.
7. Mostre que o dual de uma triangulação de um polígono simples é uma árvore, isto é, um grafo conexo sem ciclos. Conclua que se $n \geq 4$, então toda triangulação de um polígono simples de n lados tem pelo menos *duas* orelhas cujos interiores são disjuntos.

9. Triangulação de pontos

Vamos considerar agora o seguinte problema:

> *Dados n pontos no plano, decompor o seu fecho convexo em um complexo simplicial.*



Um *complexo simplicial* é uma coleção de triângulos que só se tocam em vértices ou arestas completas. Mais precisamente, dados quaisquer dois triângulos do complexo simplicial, uma das quatro situações ocorre: eles são disjuntos, eles têm exatamente um vértice em comum, eles têm exatamente uma aresta em comum, eles são iguais.

Quando os pontos são colineares, um complexo simplicial é uma coleção de arestas com vértices nos pontos dados tais que duas arestas ou são disjuntas, ou têm exatamente um vértice em comum, ou são iguais. Portanto, nesse caso, só existe uma única "triangulação": a obtida ligando os pontos na ordem em que eles ocorrem na reta que os contém.

Resolver computacionalmente o problema de triangulação de pontos é dar a estrutura topológica completa da triangulação, isto é, fornecer todas as adjacências ou pelo menos informação suficiente para extraí-las eficientemente.

Note que, em geral, um conjunto de pontos pode ter muitas triangulações diferentes.

Motivação. Usaremos como motivação o problema de interpolação de dados esparsos. É dado um conjunto S de n amostras de alturas de um terreno: (x_i, y_i, z_i) com $z_i = f(x_i, y_i)$. Vamos usar essas amostras para estimar a altura de outros pontos do terreno. Seja $p = (x, y) \in \mathbb{R}^2$ um ponto para o qual queremos estimar a

altura $f(p)$. Claro, queremos obter essa estimativa $\hat{z}$ a partir das amostras dadas.

Supondo que o terreno é contínuo, é razoável esperar que a altura $f(p)$ dependa das alturas dos pontos amostrais que estão perto de p. Além disso, também é razoável esperar que a influência de cada ponto amostral na estimativa de $f(p)$ dependa da distância até p. A área de influência do conjunto completo de amostras S é naturalmente o seu fecho convexo. Portanto, vamos nos restringir aos pontos $p \in \text{conv}(S)$. (Se $p \notin \text{conv}(S)$, então temos o problema de extrapolação e não de interpolação.)

Se temos uma triangulação de S, então todo ponto de $\text{conv}(S)$ está em algum triângulo dessa triangulação. Podemos então usar as alturas dos vértices desse triângulo para estimar $f(p)$. Mais precisamente, suponha que p está no triângulo $q_1q_2q_3$ com $q_i \in S$. Sejam λ_1, λ_2 e λ_3 as coordenadas baricêntricas de p em relação a q_1, q_2 e q_3. Então podemos tomar a estimativa

$$f(p) \approx \hat{z} = \lambda_1 z_1 + \lambda_2 z_2 + \lambda_3 z_3,$$

onde $z_i = f(q_i)$.

Dessa forma, estamos fazendo uma interpolação linear por partes das alturas amostradas, obtendo uma aproximação poliedral para o terreno. Note que essa aproximação é contínua, pois se p está numa aresta comum a dois triângulos então a altura estimada para p só depende das alturas dos vértices da aresta.

É instrutivo considerar esse mesmo problema em dimensão 1. Nesse caso, a interpolação linear por partes é feita ordenando S em relação à coordenada x e localizando cada ponto a ser interpolado na lista ordenada dos pontos amostrais. Mais precisamente, o ponto p a ser interpolado está em algum intervalo $[q_i, q_{i+1}]$ de pontos amostrais consecutivos. Nesse intervalo, aproximamos f por uma função afim que interpola as amostras:

$$\hat{f}(x) = y_i + \frac{x - x_i}{x_{i+1} - x_i}(y_{i+1} - y_i).$$

Essa expressão coincide com a estimativa baricêntrica, pois

$$x = (1-\lambda)x_i + \lambda x_{i+1} \quad \text{com} \quad \lambda = \frac{x - x_i}{x_{i+1} - x_i}.$$

Sendo assim, triangular é um tipo de ordenação em dimensões superiores.

Existência. Provemos que sempre é possível triangular um conjunto de pontos no plano. Mais precisamente,

> *Todo conjunto finito de pontos no plano pode ser triangulado.*

Já vimos que, se os pontos são colineares, então existe uma triangulação (e só uma). Suponhamos então que os pontos não são colineares. Vamos provar a existência de triangulação por indução no número de pontos. Como os pontos não são colineares, o seu fecho convexo é um polígono convexo não degenerado. Esse polígono é facilmente triangulado por meio de diagonais. (Como vimos, todos os polígonos podem ser triangulados; os polígonos convexos são especialmente fáceis de triangular.) Adicione os pontos internos ao fecho convexo um de cada vez. Cada ponto cai no interior de algum triângulo ou no interior de alguma aresta. Se ele cair no interior de um triângulo, esse triângulo é divido em três, ligando o ponto aos vértices do triângulo. Se ele cair no interior de uma aresta, os triângulos adjacentes a ela são divididos em dois, ligando o ponto aos vértices opostos à aresta nos triângulos adjacentes. Após termos adicionados todos os pontos internos ao fecho convexo, o teremos decomposto em triângulos, isto é, construído uma triangulação dos pontos dados. □

A prova acima nos dá um algoritmo para construir uma triangulação. O passo crucial nesse algoritmo é localizar cada ponto em relação à triangulação existente, pois a sua inclusão é feita em tempo constante. Fazendo a localização por meio de uma busca linear, obtemos um algoritmo quadrático. Usando uma estrutura de dados mais sofisticada, é possível fazer a localização em tempo logarítmico, obtendo um algoritmo $O(n \log n)$.

Tamanho de triangulações. Quando os pontos são colineares, só existe uma única triangulação, mas em geral um conjunto de pontos pode ter muitas triangulações diferentes. A escolha de uma

triangulação depende da aplicação. Para o problema de interpolação linear por partes, é desejável por razões numéricas que os triângulos não sejam muito finos, isto é, que não tenham ângulos muito grandes nem muito pequenos. Veremos mais adiante uma triangulação que tem boas propriedades geométricas como essa.

Embora uma conjunto de pontos possa ter muitas triangulações diferentes, todas elas têm o mesmo tamanho:

> *Seja* S *um conjunto de* n *pontos no plano, com* $n \geq 3$. *Então toda triangulação de* S *tem exatamente* $2(n-1) - h$ *triângulos e* $3(n-1) - h$ *arestas, onde* h *é o número de pontos de* S *na fronteira de* $\text{conv}(S)$.

Considere uma triangulação de S com T triângulos. Somemos todos os ângulos da triangulação. Por um lado, temos πT, pois a soma dos ângulos em cada um dos T triângulos é π. Por outro lado, considere a soma dos ângulos em torno de cada ponto de S. Para os pontos internos a $\text{conv}(S)$, essa soma é 2π para cada ponto, o que dá uma soma total de $2\pi(n-h)$. Para os pontos na fronteira de $\text{conv}(S)$, a soma total é $\pi(h-2)$, pois a fronteira é um polígono de h lados. Portanto,

$$\pi T = 2\pi(n-h) + \pi(h-2),$$

donde

$$T = 2(n-h) + (h-2) = 2(n-1) - h.$$

Seja agora A o número de arestas na triangulação. As arestas de $\text{conv}(S)$ pertencem somente a um triângulo; as arestas internas pertencem a dois triângulos. Como cada triângulo tem três arestas, se contarmos as arestas pelos triângulos cada aresta interna será contada duas vezes e cada aresta da fronteira somente uma vez. Portanto,

$$3T = 2A - h.$$

Assim,

$$2A = 3T + h = 3(2(n-1) - h) + h = 6(n-1) - 2h,$$

donde

$$A = 3(n-1) - h.$$

□

Note que h é o número de pontos de S na fronteira de $\text{conv}(S)$, e não o número de pontos extremos de S. Em outras palavras, permitimos ângulos de $180°$ na descrição de $\text{conv}(S)$.

Note ainda que, em termos de complexidade, o teorema acima diz que o tamanho de qualquer triangulação de n pontos é $\Theta(n)$.

Complexidade. Já vimos que o problema de triangulação tem solução algorítmica em tempo $O(n^2)$. Vejamos um limite inferior não trivial para esse problema. Esse limite confirma a nossa intuição de que triangular é um tipo de ordenação:

> *Todo algoritmo que triangule* n *pontos no plano leva tempo* $\Omega(n \log n)$.

De fato, vamos mostrar que é possível reduzir ORDENAÇÃO a TRIANGULAÇÃO. Sejam então dados n números que queremos ordenar: $x_1, \ldots, x_n \in \mathbf{R}$. Considere os pontos $p_i = (x_i, 0)$ e $p_0 = (x_1, 1)$. Então o conjunto $\{p_0, p_1, \ldots, p_n\}$ admite somente uma triangulação: p_0 é um vértice de todos os triângulos e a aresta oposta a p_0 em cada triângulo é $q_i q_{i+1}$, onde $q_1, \ldots, q_n$ é a sequência dos pontos já ordenados em x.