Geometria Computacional

Tópicos Avançados em Algoritmos

Prof. Hamilton José Brumatto

Linhas e Retas

As linhas, ou retas são representadas pelos pontos `(x,y)` que são raízes da equação: $ax + by + c = 0$, assim, os valores `a`, `b` e `c` são característicos de uma linha.

As linhas podem ser construídas por 2 pontos, ou por um ponto e uma inclinação. Lembrando que existe o caso exclusivo de ser uma linha vertical, neste caso, representado por $ax + c = 0$. Ou seja, $b = 0$.

Para construir uma linha a partir de dois pontos, se a diferença entre as abscissas de ambos pontos for menor que `EPS`, então é uma reta vertical, caso contrário, $a = m = \dfrac{p1.x - p2.x}{p1.y - p2.y}$ a inclinação da linha, b arbitrariamente é 1, e c o valor que depende de a, b, x e y, que representa o ponto em que a reta corta o eixo y (negativo).

Para construir uma linha a partir de um ponto e a inclinação, é direto, não dá para representar retas verticais, pois a inclinação seria infinita.

As funções que definimos para retas são:

Verificar se duas retas são paralelas:
`bool paralela(const linha& l1, const linha& l2)`
A inclinação deve ser a mesma: $l1.a == l2.a$, e também o parâmetro b, o que pode diferir é c.

Verificar se duas retas estão na mesma posição:
`bool iguais(const linha& l1, const linha& l2)`
Neste caso, todos parâmetros precisam ser iguais.

Verificar se duas retas se intercedem, e em qual ponto:
`bool intersecao(const linha &l1, const linha &l2, ponto &p)`
Precisamos resolver o sistema linear formado pelas duas retas:

$$\begin{cases} l1.a \times x + l1.b \times y + l1.c &= 0 \\ l2.a \times x + l2.b \times y + l2.c &= 0 \end{cases}$$

Este sistema só tem solução se as retas não forem paralelas (por isso há o retorno de *bool*, também é preciso o cuidado de não dividir por zero se a segunda reta for vertical.

O ângulo (representado por sua tangente) formado por duas retas em sua interseção:
`bool interangulo(const linha &l1, const linha &l2, double& angulo)`
$\tan(\theta) = \dfrac{a_1 b_2 - a_2 b_1}{a_1 a_2 + b_1 b_2}$ Da mesma forma, cuidado com retas paralelas.

O ponto mais próximo, em uma reta, de um ponto no plano. Este ponto estará em uma perpendicular à reta. Então, tendo a reta perpendicular, dada pela $m' = \dfrac{1}{m}$ e pelo ponto do plano, encontramos o ponto mais próximo a partir desta interseção.
`ponto pontoProximo(const ponto& p, const linha& l)`

Também a distância do ponto à reta, que é a distância do ponto ao ponto próximo.
`double distPlinha(const ponto& p, const linha& l)`

Segmentos
Segmentos são intervalos de reta delimitado por dois pontos, logo pode ser representada pelos dois pontos extremos.

Para os segmentos são definidas as funções:

`bool pontoRet(const ponto &p, const segmento &s);`
Esta função verifica se um ponto está no retângulo cuja diagonal é o segmento, funciona, também, se o segmento for horizontal ou vertical, ou apenas um ponto.

```
template<class ponto>
bool segIntersec(const segmento &s1, const segmento &s2, ponto &p);
```
Esta função verifica se dois segmentos intersecionam, caso positivo indica o ponto.

```
template<class ponto>
double distPseg(const segmento &s, const ponto &p)
```
Esta função retorna a menor distância de um ponto a um segmento.

```
enum{ESQ,ANTES,INI,MEIO,FIM,DEPOIS,DIR};
template<class ponto>
int positionPseg(const segmento &s, const ponto &p, bool dir=false);
```

Esta função retorna a posição de um ponto com relação a um segmento:
O valor retornado é um inteiro, de acordo com a enumeration:
0: está à esquerda do segmento.
1: está na mesma linha do segmento, antes do primeiro ponto.
2: está no primeiro ponto.
3: está na no próprio segmento, entre o primeiro e o segundo ponto.
4: está no segundo ponto.
5: está na mesma linha do segmento, após o segundo ponto.
6: está à direita do segmento.
A precisão da posição do ponto é menor que 10^{-9} para questões de arredondamento. A variável `dir` quando tornada `true` representa que a ordem dos pontos importa, caso contrário é considerado o primeiro ponto aquele que estiver mais à esquerda (no caso de reta vertical, em baixo).

Alguns problemas interessantes no URI:
URI - 1834 - Vogons, aqui podemos tratar com pontos representados por inteiros, mesmo o ângulo para saber se é esquerdo ou direito, será usado a versão de inteiros (sem o sqrt). Assim a única função importante é o positionPseg, e usando inteiros, não precisamos da comparação com EPS, ao invés de ver fabs da diferença, comparamos por igualdade. A única coisa double é a distância.
URI - 1468 - Balão.

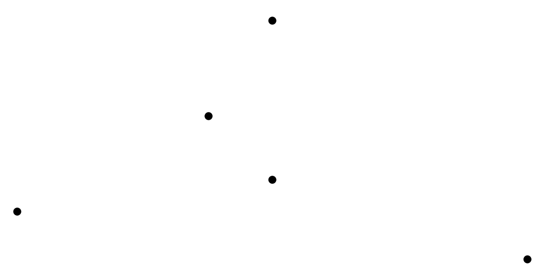
Envoltória Convexa - Um problema de Backtracking

O problema: Dado um conjunto de pontos no plano, construir o menor polígono que contenha todos os pontos.

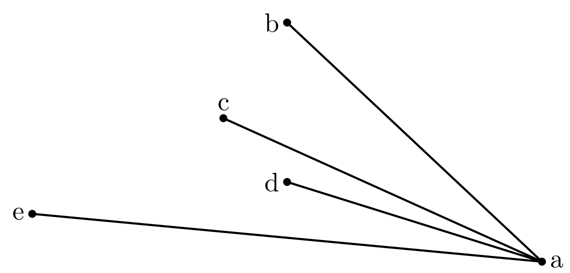
- Uma forma de interpretar a solução seria colocar uma fita elástica no entorno dos pontos, a envoltória fica determinada pelos pontos que formam os vértices.
- Para a solução por Backtracking deve-se considerar o ponto com o maior valor para a coordenada x. Deve-se ordenar os demais pontos em ângulos crescentes com relação ao ponto extremo (ou seja, decrescente em y).
- Fazer uma varredura nos pontos de acordo com a ordem criada, adicionando cada um como uma solução parcial e fazendo um backtracking para atingir um determinado ponto se algum ângulo for superior a 180^o (possui um valor negativo na rotina que calcula o ângulo).

Ilustrando o algoritmo

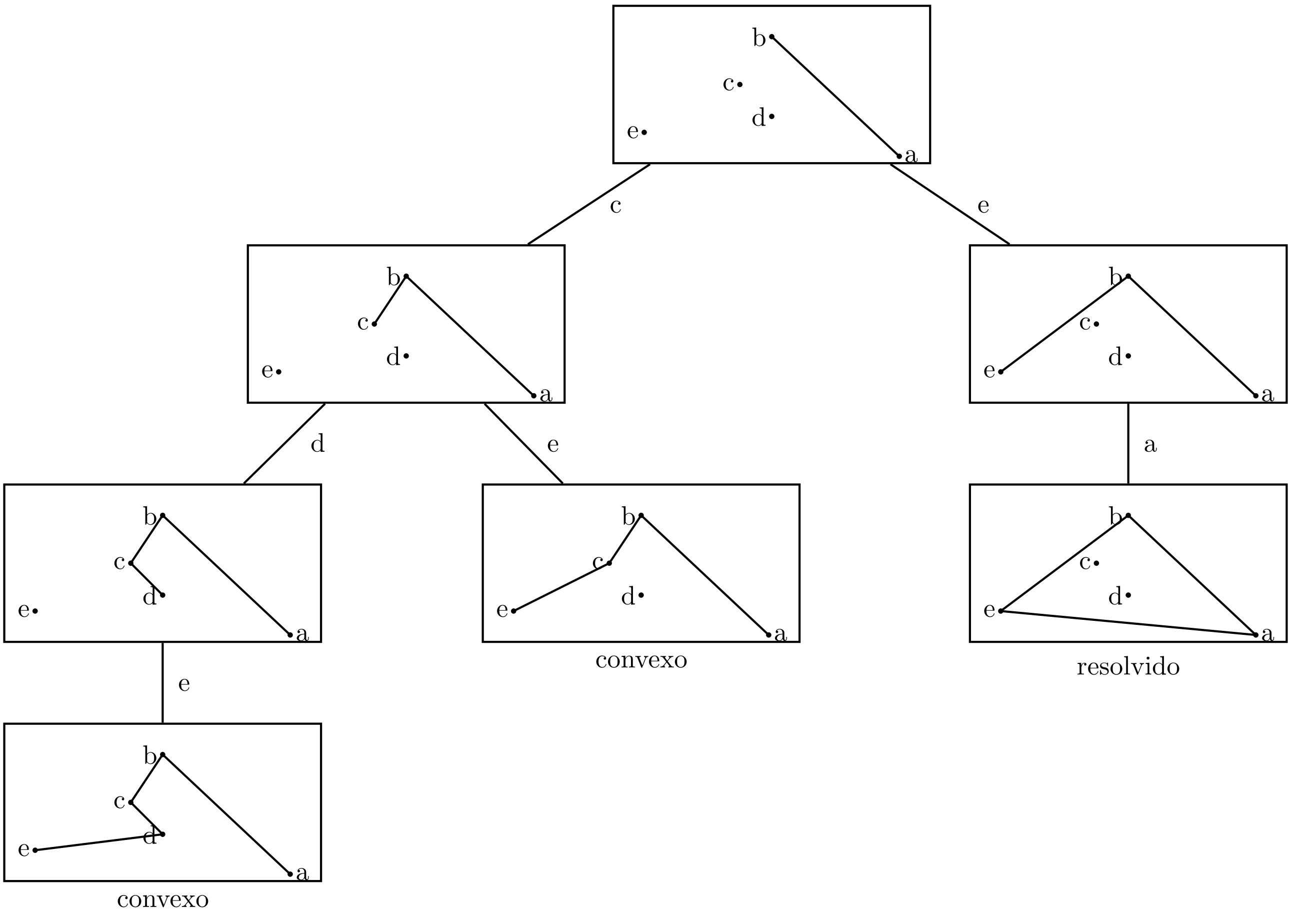
- vamos considerar os seguintes pontos



- A ordenação resulta em:



Aplicando o algoritmo no exemplo temos a árvore de busca do backtracking



Problemas de Envoltória Convexa:

URI - 1336 - Cerca do Jardim
URI - 1464 - Camadas de Cebola
URI - 1982 - Novos Computadores