tutorial.md

Tutorial TEP 02/2016 - Prova 01

Acerte a lata

Categoria: *Geometria Computacional* **Tópico Principal**: *Distância entre ponto e segmento de reta* **Dificuldade**: Médio

A ideia central deste problema é computar a distância entre um ponto *Q* (onde está localizada a lata) e o segmento de reta formado pelos pontos L, P, que marcam o lançamento e o ponto de parada.

Esta distância pode ser computada determinando o ponto *R* da reta que passa por *L, P* que é o mais próximo possível de *Q*. Se este ponto estiver no segmento, a distância será *d = dist(Q, R)*; caso contrário, *d = min(dist(L, R), dist(L, P))*.

A classe abaixo representa uma reta e contém o método que computa o ponto *R* mais próximo possível de *Q*.

```
class Line {
public:
    double a;
    double b;
    double c;

    Line(const Point& p, const Point& q)
    {
        a = p.y - q.y;
        b = q.x - p.x;
        c = p.x * q.y - p.y * q.x;
    }

    Point closest(const Point& Q) const
    {
        auto m = a*a + b*b;
        auto x = (b*(b*Q.x - a*Q.y) - a*c)/m;
        auto y = (a*(-b*Q.x + a*Q.y) - b*c)/m;

        return Point(x, y);
    }
};
```

Já a classe abaixo representa um segmento, com um método que determina se um ponto *P*, que pertence à reta que passa pelo segmento *AB*, está contido ou não no segmento.

```
class Segment {
public:

    Point A, B;

    Segment(const Point& Av, const Point& Bv) : A(Av), B(Bv) {}

    bool contains(const Point& P) const
    {
        if (A.x == B.x)
            return min(A.y, B.y) <= P.y and P.y <= max(A.y, B.y);
        else
            return min(A.x, B.x) <= P.x and P.x <= max(A.x, B.x);
    }
```

```
};
```

Com estas informações, basta seguir os critérios de vitória e desempate listados no texto do problema.

Bom trabalho

Categoria: *Geometria Computacional* **Tópico Principal**: *Orientação de um ponto em relação a uma reta* **Dificuldade**: Fácil

Neste problema, dado um ponto *R* e uma reta *r* que passa pelos pontos *P* e *Q*, é preciso determinar qual dos três casos abaixo é verdadeiro:

1. *R* pertence a *r*;
2. *R* está à esquerda de *r*, no sentido de *P* a *Q*;
3. *R* está à direita de *r*, no sentido de *P* a *Q*.

Isto pode ser feito através do discriminante *D*, implementado abaixo:

```
int D(const Point& P, const Point& Q, const Point& R)
{
    return (P.x * Q.y + P.y * R.x + Q.x * R.y) - (R.x * Q.y + R.y * P.x + Q.x * P.y);
}
```

Assim, basta incrementar o contador de estacas desalinhas sempre que *D* for diferente de zero, e computador o custo final através de uma multiplicação simples. Uma forma de evitar erros de arredondamento é trabalhar com valores em centavos, fazendo a conversão apenas na hora da impressão.

Para ler o valor como centavos, basta fazer

```
int R, C;
scanf("%d,%d", &R, &C);
int cents = R*100 + C;
```

A impressão do custo pode ser feita através da expressão

```
printf("Custo: R$ %d,%02d\n", cost / 100, cost % 100);
```

Cães e Gatos

Categoria: *Geometria Computacional* **Tópico Principal**: *Vetores* **Dificuldade**: Médio

Para cada um dos cães, a primeira providência é computar a distância entre o ponto *C* onde sua corrente está presa e o gato (ponto *G*). Se esta distância *d* for menor ou igual ao comprimento da *M* corrente, o gato perderá uma de suas vidas.

Caso *d* > *M*, o cachoro correrá da direção do gato, até a onde sua corrente permitir, ficando o mais próximo possível do felino. Em termos geométricos, o cão ficará posicionado em um ponto *P* da reta *CG*, ficando a uma distância *M* de *C*.

Para encontrar *P*, basta computar o vetor *v* que parte de *C* a *G* (*v* = *G* - *C*) e, em seguida, o vetor unitário *u* que tem mesma direção e sentido de *v* (*u* = *v*/|*v*|). Assim, P = C + u (adição e subtração de pontos: coordenada a coordenada).

Vale notar que, pela restrição da entrada, restará ao gato sempre ao menos uma vida, e que a primeira comparação (entre *d* e *M*) pode ser feita sem o uso de ponto flutuante (compare d^2 com M^2).

Desenvolvendo a API

Categoria: *Geometria Computacional* **Tópico Principal**: *Rotações e escala* **Dificuldade**: Fácil

O problema consiste em aplicar rotações e operações de escala em um dado ponto. Ele se torna um pouco mais fácil do que o problema geral por dois motivos:

1. As rotações são feitas apenas em múltiplos de 90º;
2. As constantes de escala são potências positivas e negativas de 2.

Estes fatores permitem trabalhar com coordenadas inteiras *x, y* e um fator de zoom *k*, também inteiro. Rotacionar 90º, no sentido anti-horário, é equivalente a permutar as coordenadas do ponto, tomando o simétrico da primeira nova coordenada:

```
using ii = pair<long long, long long>;

ii rotate90(const ii& v)
{
    return ii(-v.y, v.x);
}
```

As operações de *zoom out* e *zoom in* equivalem a incrementos e decrementos do fator *k*. As operações só precisam ser efetivamente realizadas no momento da impressão, tomando cuidado para não alterar os valores de *x, y*: use duas variáveis em ponto flutuante para as operações caso *k < 0*, e use o operator `<<` caso *k > 0*.

O problema se tornaria mais difícil se removida a restrição do módulo das coordenadas, mas a abordagem de acumular o fator de *zoom* ainda seria válida.