Programação Estruturada

Aula 11 - Jogo do labirinto parte 2 – recursão

Apresentação

Na aula passada, você viu como programar os recursos básicos do jogo do labirinto usando o console (monitor). Agora que já temos um tabuleiro sendo mostrado na tela, vamos implementar um algoritmo para que o computador encontre sozinho uma solução de caminho para o labirinto mostrado.

Para implementação dessa solução, iremos utilizar a recursão. Como você verá, essa forma de implementação simplifica o desenvolvimento de programas como o mostrado nessa aula.

Faça uma boa leitura e uma boa programação do jogo!

Vídeo 01 - Apresentação

Objetivos

Ao final desta aula, você será capaz de:

- Usar a recursão na construção de programas.
- Implementar um jogo de labirinto no qual o próprio computador aprende o caminho a ser seguido.

1. Introdução

Vamos agora dar continuidade ao desenvolvimento do jogo de labirinto, cujos recursos básicos foram construídos na aula anterior.

Você lembra da classe Labirinto, que possui procedimentos como “inicializarMatriz()” e “imprimir()”? E que o tabuleiro do jogo está sendo representado por uma matriz de caracteres? Se você não se lembra o que esses procedimentos fazem, volte à aula anterior e olhe novamente o código do labirinto, pois a aula de hoje irá estender esse programa com novas funcionalidades.

Nesta aula, você irá implementar funções de busca por caminhos no labirinto. A implementação dessa função pode ser feita de várias formas, como por exemplo, a recursão. Mas antes disso, precisamos definir quais são os pontos de saída e de chegada ao labirinto.

2. Pontos de Saída e de Chegada

O tabuleiro que foi implementado na aula passada possui paredes externas que delimitam o tabuleiro, bem como paredes internas, que representam os obstáculos dentro do labirinto. Esses obstáculos foram posicionados de forma aleatória, certo?

Pois bem, precisamos agora definir um ponto de saída e um ponto de chegada para que seja possível ao computador tentar encontrar um caminho válido entre esses dois pontos, respeitando, é claro, todos os obstáculos do labirinto. Para isso, precisamos definir duas novas constantes:

```
1 private static final char INICIO = 'I';
2 private static final char DESTINO = 'D';
```

Essas constantes representam os caracteres utilizados para representar o ponto de início e de destino (chegada) em questão. Elas devem ser definidas juntas com as outras constantes já definidas na aula anterior (PAREDE_VERTICAL, PAREDE_HORIZONTAL etc.).

Agora que já temos os caracteres a serem utilizados para definir as posições de início e destino no labirinto, precisamos escolher os locais no tabuleiro onde iremos colocar esses caracteres. Vamos começar com a posição de início. Além da constante INICIO, vamos definir duas variáveis para representar a posição no tabuleiro desse caractere de início:

```
1 private static int linhaInicio;
2 private static int colunaInicio;
```

Note que não estamos usando a palavra-chave final na declaração das variáveis, pois não nos interessa declará-las como constantes. Isso porque você pode querer alterar o programa para jogar mais de uma vez sem ter que reiniciar o programa. Nesse caso, o valor da linha e da coluna da matriz que representa o início do jogo mudaria, não podendo assim as variáveis ser declaradas como constantes.

Para definir os valores dessas variáveis, vamos primeiro definir uma função que gera um número inteiro entre um dado intervalo:

```
1 public static int gerarNumero(int minimo, int maximo) {
2    int valor = (int) Math.round(Math.random()  * (maximo - minimo));
3    return minimo + valor;
4 }
```

Note que essa função recebe como parâmetro dois números inteiros, representando os valores mínimo e máximo que a função deve retornar. Para isso, usamos a função “Math.random()” que gera um número aleatório entre 0 e 1. Esse número é então multiplicado pela diferença entre o valor máximo e o mínimo, que basicamente é a amplitude do intervalo.

Por exemplo, se a função receber os números 10 e 15 como parâmetro, a amplitude do intervalo será de 5. Quando multiplicado por um número real (*float*) entre 0 e 1, o resultado é um valor do tipo *float* entre 0 e 5. A função “Math.round()” é então utilizada para arredondar esse número, ou seja, transformá-lo novamente em um inteiro. O uso de (*int*) indica que o inteiro retornado que esperamos é do tipo *int*, e não do tipo *long*, tipo padrão de retorno da função “Math.round()”. Por fim, o valor entre 0 e 5 é adicionado ao número mínimo do intervalo, que nesse caso é o valor 10. No final, o resultado dessa operação será algo entre 10 e 15, percebeu?

Agora que temos essa função, podemos colocar no final da função “inicializarMatriz()” o seguinte código:

```
linhaInicio = gerarNumero(1, TAMANHO / 2 - 1);
colunaInicio = gerarNumero(1, TAMANHO / 2 - 1);
tabuleiro[linhaInicio][colunaInicio] = INICIO;
```

Isso fará com que o caractere que representa o início do labirinto seja posicionado em uma linha e coluna cujos índices estão entre 1 (não usamos o zero porque é a parede externa) e a metade do tabuleiro (*TAMANHO* / 2 - 1). Isso quer dizer que o ponto de início será posicionado na parte superior esquerda do tabuleiro!

Podemos fazer a mesma coisa para a posição do caractere de destino do labirinto, só que posicionando-o na parte inferior à direita do tabuleiro:

```
int linhaDestino = gerarNumero(TAMANHO / 2, TAMANHO - 2);
int colunaDestino = gerarNumero(TAMANHO / 2, TAMANHO - 2);
tabuleiro[linhaDestino][colunaDestino] = DESTINO;
```

O código mostrado deve ser posicionado também no final do procedimento “inicializarMatriz()”. Note que deve haver a declaração de duas variáveis auxiliares usadas para calcular a linha e coluna do caractere que representa o destino do labirinto.

Vídeo 02 - Pontos Início e Destino

Atividade 01

1. Altere sua implementação do programa Labirinto para apresentar o tabuleiro com as posições de início e destino, como mostrado nesta parte da aula. Tente não copiar o código mostrado, mas fazer sem olhar o material da aula. Relate se você conseguiu fazer a implementação sem copiar, se você fez algo de diferente em termos de implementação ou se a sua implementação ficou igual à mostrada na aula.
2. Execute várias vezes o programa Labirinto e relate se houve mudanças na posição dos pontos de início e destino do labirinto.
3. Teste o uso de outros caracteres para as constantes de INICIO e DESTINO usadas no programa e indique se você encontrou valores mais adequados (para deixar o tabuleiro mais bonito, mais simples de ser visualizado etc. que os mostrados na aula).
4. Altere a função "inicializarMatriz()" para posicionar o caractere de início na parte inferior à esquerda do tabuleiro, e para posicionar o caractere de destino na parte superior à direita do tabuleiro. Execute e confira se sua implementação está correta.

3. Busca do Caminho do Labirinto

Chegou a hora de implementar a “inteligência” do jogo, na qual o computador irá identificar de forma automática um caminho que liga o ponto de início ao destino no tabuleiro. Para fazer isso, vamos definir duas novas constantes:

```
1 private static final char CAMINHO = '.';
2 private static final char SEM_SAIDA = 'x';
```

A constante de nome CAMINHO representa o caractere que marca o caminho por onde o computador está percorrendo. Já a constante de nome SEM_SAIDA será usada para marcar os caminhos pelos quais o computador já percorreu e que verificou que por lá não é possível alcançar o destino.

Vamos agora à função de busca pelo caminho. Para isso definimos a função “procurarCaminho()”, que recebe dois parâmetros do tipo inteiro, a linha e a coluna de início do caminho. Tente entender o que o código dessa função faz:

```
1  public static boolean procurarCaminho(int linhaAtual, int colunaAtual) {
2      int proxLinha;
3      int proxColuna;
4      boolean achou = false;
5      // tenta subir
6      proxLinha = linhaAtual - 1;
7      proxColuna = colunaAtual;
8      achou = tentarCaminho(proxLinha, proxColuna);
9      // tenta descer
10     if (!achou) {
11         proxLinha = linhaAtual + 1;
12         proxColuna = colunaAtual;
13         achou = tentarCaminho(proxLinha, proxColuna);
14     }
15     // tenta à esquerda
16     if (!achou) {
17         proxLinha = linhaAtual;
18         proxColuna = colunaAtual - 1;
19         achou = tentarCaminho(proxLinha, proxColuna);
20     }
21     // tenta à direita
22     if (!achou) {
23         proxLinha = linhaAtual;
24         proxColuna = colunaAtual + 1;
25         achou = tentarCaminho(proxLinha, proxColuna);
26     }
27     return achou;
28 }
```

Note que essa função retorna um boleano de valor *true*, caso um caminho tenha sido encontrado; e *false*, caso contrário. Seu funcionamento é basicamente o seguinte. O algoritmo da função tenta seguir o caminho subindo no tabuleiro (manter coluna, tentando linha anterior), se não der ele tenta descendo (manter coluna e tentar linha posterior), se não der vai pela esquerda (manter linha e tentar coluna anterior) e por último tenta pela direita (manter linha e tentar coluna posterior). Para realizar essas tentativas, é usada a função “tentarCaminho()”, cujo código será mostrado a seguir. Essa nova função recebe como parâmetro a posição do caminho que se quer tentar percorrer.

```
private static boolean tentarCaminho(int proxLinha, int proxColuna) {
   boolean achou = false;
   if (tabuleiro[proxLinha][proxColuna] == DESTINO) {
      achou = true;
   } else if (posicaoVazia(proxLinha, proxColuna)) {
      // marcar
      tabuleiro[proxLinha][proxColuna] = CAMINHO;
      imprimir();
      achou = procurarCaminho(proxLinha, proxColuna);
      if (!achou) {
         tabuleiro[proxLinha][proxColuna] = SEM_SAIDA;
         imprimir();
      }
   }
   return achou;
}
```

A função “tentarCaminho()” basicamente realiza vários testes. O primeiro deles é se a posição que se está tentando caminhar é a de destino (*tabuleiro*[proxLinha][proxColuna] == *DESTINO*). Se for esse o caso, então é porque o destino foi alcançado e retorna ao valor *true*. Caso contrário, verifica-se se a posição é válida para caminhar através de uma função “posicaoVazia()”, mostrada mais adiante e que retorna *true* apenas quando a posição passada como parâmetro estiver vazia, ou seja, é um posição válida para formar um caminho no labirinto (está no tabuleiro e não é uma parede ou obstáculo).

Nesse caso, o caminho é marcado (posição preenchida com o valor da constante CAMINHO) e imprime-se novamente na tela o tabuleiro com os seguintes comandos:

```
tabuleiro[proxLinha][proxColuna] = CAMINHO;
imprimir();
```

A ideia é então continuar a busca a partir dessa posição marcada. Isto é feito chamando-se novamente a função “procurarCaminho()”. Note que essa função recebe como parâmetro as novas posições “proxLinha” e “proxColuna”., indicando que será realizada a busca a partir dessa nova posição:

```
achou = procurarCaminho(proxLinha, proxColuna);
```

Nesse momento é que entra a recursão! O processo mostrado irá se repetir a partir dessa “nova posição inicial”. Caso a função “procurarCaminho()” retorne *true*, isto indicará que foi possível encontrar um caminho até o destino a partir daquela

posição “proxLinha” e “proxColuna”. Caso retorne *false*, quer dizer que não foi possível encontrar o caminho, então devemos descartar aquela posição. Isto é feito justamente pelos seguintes comandos:

```
if (!achou) {
    tabuleiro[proxLinha][proxColuna] = SEM_SAIDA;
    imprimir();
}
```

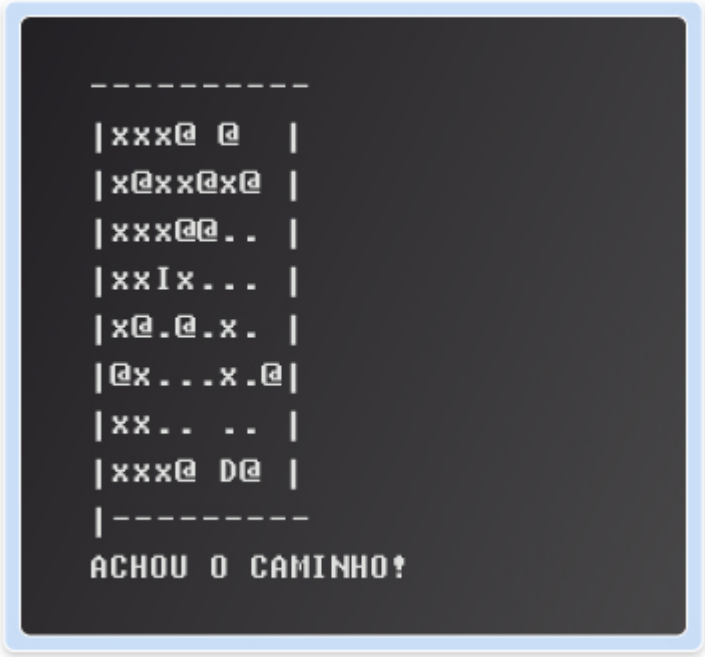
Por fim, a função “tentarCaminho()” retorna ao valor da variável “achou”, indicando se foi possível ou não achar um caminho dentro do tabuleiro. Veja agora o código da função “posicaoVazia()”, utilizado para verificar se uma determinada posição pode ser usada em um caminho dentro do tabuleiro do labirinto:

```
public static boolean posicaoVazia(int linha, int coluna) {
    boolean vazio = false;
    if (linha >= 0 && coluna >= 0 && linha < TAMANHO && coluna < TAMANHO) {
        vazio = (tabuleiro[linha][coluna] == VAZIO);
    }
    return vazio;
}
```

Basicamente, o que essa função faz é verificar se a posição linha e coluna passada como parâmetro está dentro do tabuleiro e se está vazia, podendo ser utilizada em um labirinto. Para esse código poder ser utilizado, precisamos chamá-lo no final do corpo da função “main()”, adicionando os seguintes comandos:

```
System.out.println("\n- Procurando solução -\n");
    boolean achou = procurarCaminho(linhaInicio, colunaInicio);
    if (achou) {
        System.out.println("ACHOU O CAMINHO!");
    } else {
        System.out.println("Não tem caminho!");
}
```

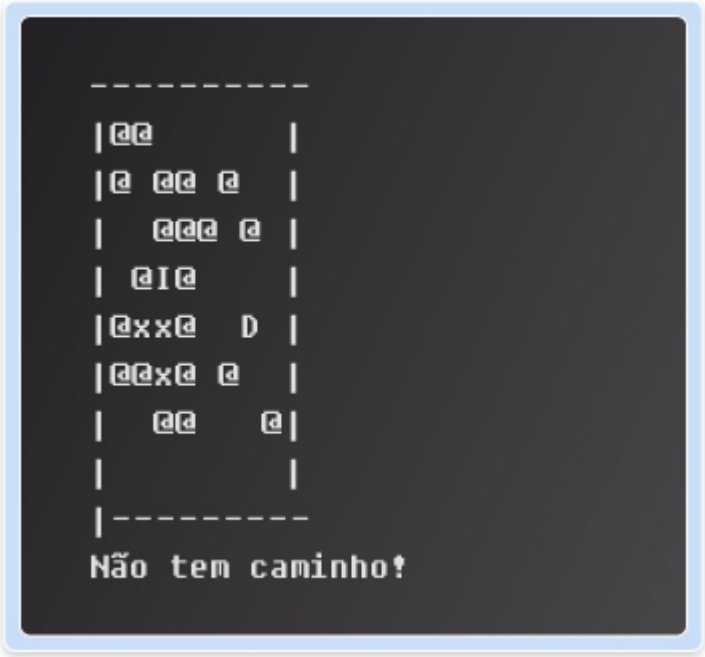
Esse código chamará a função “procurarCaminho()” a partir da linha e coluna inicial. Lembre-se que essa função vai percorrendo o tabuleiro, marcando o caminho encontrado com os caracteres ‘.’, os caminhos que não levaram a nada preenche-se com ‘x’ e imprime-se o tabuleiro cada vez que ele é modificado. Um possível tabuleiro final é o seguinte:



As posições do tabuleiro preenchidas com ‘x’ indicam posições não percorridas. Já os pontos seguem da posição de início ‘I’ até a posição de destino ‘D’. Note que o caminho não é ótimo, ele vai tentando na sorte. Algoritmos mais inteligentes poderiam ser utilizados para encontrar o caminho ótimo, mas isso está fora do escopo desta aula.

Vídeo 03 - buscaCaminho

Além disso, podemos ter saídas como a mostrada a seguir, onde não há caminho válido:



Para ficar mais fácil de entender o caminho percorrido pelo algoritmo, você pode adicionar o seguinte código ao final da função “imprimir”. Esse código é responsável por suspender a execução do programa pelo tempo indicado em “milissegundos” (no caso, 300ms).

```
try {
    Thread.sleep(300);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

O bloco try-catch recupera exceções que ocorram na chamada de uma função. A função printStackTrace() retorna a lista de erros que foram gerados a partir de uma exceção. Os detalhes sobre esses dois elementos serão estudados em Programação Orientada a Objetos. Por enquanto, basta lembrar de colocá-los quando o programa solicitar!

Vídeo 04 - buscaCaminho delay

Atividade 02

1. Altere sua implementação do programa Labirinto para ele procurar e mostrar um caminho válido, como mostrado nesta última parte da aula. Tente não copiar o código mostrado, mas fazer sem olhar o material da aula. Relate se você conseguiu fazer a implementação sem copiar, se você fez algo de diferente em termos de implementação ou se a sua implementação ficou igual à mostrada na aula.

2. Execute várias vezes o programa Labirinto e relate se houve mudanças na posição dos pontos de início e destino e nos caminhos encontrados. Eles foram os menores caminhos possíveis dentro do tabuleiro? Sempre foi possível encontrar caminhos válidos?

3. Teste o uso de outros caracteres para as constantes de CAMINHO e SEM_SAIDA usadas no programa e indique se você encontrou valores mais adequados (para deixar o tabuleiro mais bonito, mais simples de ser visualizado etc.) que os mostrados na aula.

4. Altere o tempo em "milissegundo"s do comando "Thread.sleep()" para mais e para menos. Relate o que acontece com essas mudanças, e qual seria o melhor valor para você poder acompanhar a execução do algoritmo de busca.

4. Resumo

Nesta aula, você aprendeu conceitos sobre recursão aplicada à programação, que no contexto dessa matéria é a implementação de funções e procedimentos usando a própria definição da função. Nesta aula, o corpo da função “procurarCaminho()” faz uma chamada à função “tentarCaminho()” que por sua vez chama de volta a função “procurarCaminho()”, configurando assim a recursão. Esta forma de implementação simplifica o desenvolvimento de alguns programas, como o mostrado nessa aula. Ele seria bem mais complicado se implementado sem recursão.

Espero que você tenha gostado e até a próxima aula!

5. Autoavaliação

1. Qual a ideia do uso da recursão utilizada na programação do programa Labirinto.
2. Na sua opinião, códigos que utilizam recursão são intuitivos? Você conseguiu entender facilmente o código recursivo do programa Labirinto?

6. Referências

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi de. **Fundamentos da programação de computadores**: algoritmos, Pascal, C/C++ e Java. São Paulo: Editora Pearson, 2008.

THE JAVA tutorials. Disponível em: <http://download.oracle.com/javase/tutorial/>. Acesso em: 6 dez. 2011.