Refinamentos sucessivos

Objetivos:

- Estudar a técnica de refinamentos sucessivos

Jogo: 2048

- Implementaremos o jogo 2048, com o objetivo de estudar a manipulação dos elementos de uma matriz

- Utilizaremos a técnica de refinamentos sucessivos

Jogo: 2048 - Regras

- Este jogo é jogado em um tabuleiro 4x4, no qual inicialmente existem duas posições com peças de valor 2 ou 4 em posições aleatórias. As outras casas contém posições vazias

Jogo: 2048 - Regras

- As peças podem ser movimentadas para cima, para baixo, para a direita ou para esquerda

- Ao fim de cada movimento, uma peça com o valor 2 ou com o valor 4, esta com menor probabilidade, é colocada no tabuleiro, em uma posição vazia

Jogo: 2048 - Regras

- Ao fazer um movimento para alguma direção, todas as peças são movimentadas para o extremo desta direção

- Se existe um movimento válido ele deve ser feito

- Quando duas peças de igual valor se chocam, elas viram uma só peça de valor igual à soma das peças originais

Jogo: 2048 - Regras

• O objetivo é obter uma peça com o valor “2048” em alguma posição do tabuleiro, embora seja possível obter valores maiores

• O jogo mostra uma pontuação, que é a soma total das peças que foram fundidas durante o jogo

Refinamentos sucessivos

• Programar por refinamentos sucessivos é propor uma solução em alto nível e progressivamente tomar decisões sobre estruturas de dados e algoritmos

• Neste processo, procura-se deixar os detalhes para as próximas fases, mantendo código de alto nível em várias das subfases do projeto

Jogo: 2048 – Algoritmo base

- Um primeiro algoritmo em alto nível pode ser este

2048: primeiro algoritmo

```
Begin
    “Iniciar Jogo”;
    “Mostra o tabuleiro e os pontos”
    Enquanto “existe movimento possível” faça
    Begin
        “Escolhe direção do movimento”
        “Movimenta peças e atualiza os pontos”
        “Sorteia nova peça e sua posição”
        “Mostra o tabuleiro e os pontos”
    End;
End.
```

Refinando o algoritmo

- Com isto é possível estabelecer os procedimentos e funções iniciais, independentemente das estruturas de dados

- No entanto, é possível que possa haver necessidade de alguma alteração em função de alguma necessidade que venha a surgir

Refinando o algoritmo

- Procedure IniciarJogo
(var T: tabuleiro; var Pontos: integer);

Faz a inicialização das estruturas de dados e variáveis globais do programa

Refinando o algoritmo

- Function ExisteMovimento (var T: tabuleiro): boolean;

Verifica a existência de movimento possível. Retorna *true* caso algum existe e *false* caso contrário

Refinando o algoritmo

- Function EscolherDirecao: tipoDirecao;

A princípio não precisa de parâmetros, apenas faz a interface com o usuário, que informa a direção desejada do movimento: para a direita, para a esquerda, para cima ou para baixo

Retorna algum código referente ao movimento escolhido

Refinando o algoritmo

- Procedure MovimentarPeca
(var T: tabuleiro; Direcao: tipoDirecao;
 var Pontos: integer);

Com a garantia de que existe movimento possível, aplica o movimento escolhido: atualiza o tabuleiro segundo as regras do jogo, movimentando as peças na direção desejada. Retorna a pontuação obtida com o movimento. Deve garantir que um movimento ocorra

Refinando o algoritmo

- Procedure InserirNovaPeca (var T: tabuleiro);

Com a garantia de que existe posição livre, faz a escolha de uma posição aleatória que ainda esteja vazia e sorteia um valor que pode ser 2 ou 4, sendo este último valor de menor probabilidade do que o primeiro

Refinando o algoritmo

- Procedure MostrarJogoAtual
(var T: tabuleiro; Pontos: integer);

Cuida da parte visual do programa, mostrando o tabuleiro atual e a pontuação obtida até o momento

1º refinamento do algoritmo

```
Begin
    IniciarJogo (T,Pontos);
    MostrarJogoAtual (T,Pontos);
    While ExisteMovimento (T) do
    Begin
        Direcao:= EscolheDirecao;
        MovimentarPecas (T,Direcao,Pontos);
        InserirNovaPeca (T);
        MostrarJogoAtual (T,Pontos);
    End;
End.
```

Análise do 1º refinamento

- Não é possível entrar em loop infinito, pois quando se entra no laço é porque existiu movimento válido, logo no pior caso uma casa ficou vazia e ela seria escolhida

- Na próxima rodada, se o tabuleiro ficou cheio, o teste do laço identifica e aborta o laço

Estruturas de dados

- Agora podemos escolher a(s) estrutura(s) de dados para representar o jogo

- Deve-se levar em conta uma ideia dos algoritmos que serão necessários

Estruturas de dados

- No nosso caso, precisaremos movimentar peças, somar valores, etc.

- Nossa primeira decisão será a utilização de uma matriz

Estruturas de dados

• Como iremos somar peças, nada mais intuitivo do que fazer uma matriz de números inteiros, originalmente 4 x 4, mas prevista para poder ser maior, caso desejemos jogar um jogo com tabuleiro maior

• Problema: como representar as posições vazias?

Estruturas de dados

- Podemos usar o número zero para representar as posições vazias

- Para fins de efeito visual, basta trocar zeros por brancos, e o usuário terá a impressão de que as casas estão vazias

- Trivialmente, basta imprimir os valores das casas não nulas e o visual está completo

2048: primeira estrutura de dados

```
CONST
    MAX = 4;


TYPE
    tabuleiro = array [1..MAX,1..MAX] of integer;


Var
    T: tabuleiro;
     Direcao, Pontos: integer;
```

2º refinamento

- podemos detalhar melhor as funções e procedimentos que são possíveis a partir da decisão tomada com base na estrutura de dados escolhida

Detalhamento das subrotinas

- Como iniciar o jogo?

- Basta uma rotina para zerar todas as casas da matriz e sortear duas posições para conter as primeiras duas peças

- A pontuação será zerada também.

Detalhamento das subrotinas

```
Procedure IniciarJogo (var T: tabuleiro; var Pontos: integer);
Var i,j: integer;
Begin
    For i:= 1 to max do
        For j:= 1 to max do
            T[i,j]:= 0;
    For i:= 1 to 2 do
        InserirNovaPeca (T);
    Pontos:= 0;
End;
```

Detalhamento das subrotinas

- Como mostrar o jogo na tela?

- Basta imprimir a matriz, trocando zeros por espaços em branco. Para a diagramação correta usaremos o “:” do comando write

- Usaremos a procedure “clrscr” (clear screen) da biblioteca CRT para imprimir a nova matriz sobre a anterior, criando o aspecto visual de um jogo

Detalhamento das subrotinas

```
Procedure MostrarJogoAtual (var T: tabuleiro; Pontos: integer);
Var i,j: integer;
Begin
    For i:= 1 to max do
    Begin
        For j:= 1 to max do
            If T[i,j] = 0 then Write ('      ')  // 6 espacos em branco
            Else Write (T[i,j]:6 ,'  ');
        Writeln;
    End;
    Writeln ('Pontuação: ', Pontos);
End;
```

Detalhamento das subrotinas

- Como inserir uma nova peça?

- São duas ações diferentes: uma para sortear uma posição vazia e outra para sortear um valor para a nova peça

- Este procedimento é usado no programa principal e também na procedure IniciarJogo, portanto deve funcionar em ambas

Detalhamento das subrotinas

```
Procedure InserirNovaPeca (var T: tabuleiro);
Var x,y: integer;
Begin
    SortearNovaPosicao (T,x,y);
    T[x,y]:= SortearValor (MAXVALORPECA);
End;
```

A função de sorteio do valor deve se preocupar em dar maior probabilidade para sortear o valor 2 e menor probabilidade para o valor 4

Detalhamento das subrotinas

- Como sortear uma posição vazia?

- Não queremos percorrer a matriz inteira a cada novo sorteio, logo a questão é: como fazer a operação de maneira eficiente?

- O algoritmo seguinte é uma primeira tentativa

Detalhamento das subrotinas

```
Procedure SortearNovaPosicao (var T: tabuleiro; var x,y: integer);
Begin
    Repeat
        x:= random (max) + 1;
        y:= random (max) + 1;
    Until T[x,y] = 0;
End;
```

Primeiro problema

- Este algoritmo pode ser ineficiente no caso da matriz estar muito “cheia”

- Qual seria uma boa alternativa? Ela existe?

Segunda tentativa

- Podemos manter uma lista com as posições livres?

- Por exemplo um “vetor” com pares de elementos com garantia de serem posições vazias

- Quais as vantagens e desvantagens?

Vantagens

- Facilitaria sortear as posições livres, bastaria sortear um único índice deste “vetor”

- Criar esta lista é simples, basta iniciá-la com todos os elementos da matriz original, menos as duas posições originalmemente sorteadas

- Mas, e para manter a lista ao longo do jogo?

Desvantagens

- Manter esta lista pode se tornar difícil ao longo do processo de movimentação das peças

- Mas ainda não sabemos como será o movimento das peças...

Decisão

- Por enquanto vamos usar o algoritmo com o repeat

- Se resolvermos alterar isto, o restante do código não deverá sofrer alterações, pois a implementação está sendo feita de maneira bastante modular

Detalhamento das subrotinas

- Como sortear um valor para uma nova peça?

- Pelas regras do jogo, os valores sorteados devem ser 2 ou 4, mas a probabilidade de sortear o 4 deve ser menor do que sortear o 2

- O algoritmo seguinte é uma primeira tentativa

Detalhamento das subrotinas

```
CONST
    PERCENT = 5; MAXVALORPECA = 4;
Function SortearValor (MaxValor:integer ) : integer;
Begin
    SortearValor:= 2;
    If random (PERCENT) = 0 then
        SortearValor:= MaxValor;
End;
```

Detalhamento das subrotinas

- Como escolher a direção do movimento?

- Queremos que o usuário use as flechas do teclado, então teremos que usar a rotina *readkey*, da biblioteca CRT, que permite ler algo do teclado sem ter que apertar “enter”

- Definiremos os seguintes códigos:
  DIR=1 ; ESQ=2; CIMA=3; BAIXO=4;

Detalhamento das subrotinas

- Definiremos os seguintes códigos:

  DIR=1 ; ESQ=2; CIMA=3; BAIXO=4;

- Portanto, o tipoDirecao pode ser Integer

- Usaremos, para facilidade visual, um novo comando do Pascal, o comando *case*

Lendo das flechas do teclado

```
Function EscolheDirecao: integer;
Var ch: char;
begin
        ch:= readkey;
        case ch of
          #0: begin
                ch:= readkey;
                case ch of
                  #72 : escolheDirecao:= cima;      (*^*)
                  #75 : escolheDirecao:= esq;       (*<*)
                  #77 : escolheDirecao:= dir;       (*>*)
                  #80 : escolheDirecao:= baixo;     (*v*)
                end;
              end;
        End;
    End;
```

Revisão

- Voltemos ao algoritmo principal em seu estado atual

- Por onde continuaremos?

1º refinamento do algoritmo

```
Begin
    IniciarJogo (T,Pontos);
    MostrarJogoAtual (T,Pontos);
    While ExisteMovimento (T) do
    Begin
        Direcao:= EscolheDirecao;
        MovimentarPecas (T,Direcao,Pontos);
        InserirNovaPeca (T);
        MostrarJogoAtual (T,Pontos);
    End;
End.
```

Revisão

- Falta a implementação de duas rotinas:
- ExisteMovimento (T)
- MovimentarPecas (T,Direcao,Pontos)

Detalhamento das subrotinas

- Como descobrir se existe movimento válido?

- Existe movimento válido se houver pelo menos uma casa vazia

- No pior caso, se a matriz estiver completamente cheia, ainda existe um movimento se houver pelo menos duas peças vizinhas de igual valor, na horizontal ou vertical

Detalhamento das subrotinas

- Uma implementação ingênua exige passar por toda a matriz no pior caso

- Uma implementação mais eficiente exigiria outras estruturas de dados: lista com as casas vazias; localização das peças vizinhas de igual valor

- A primeira tentativa é passar por toda a matriz

Detalhamento das subrotinas

- Passar por toda a matriz é simples, basta parar quando uma casa vazia for encontrada, ou quando dois vizinhos tiverem igual valor

Modificação da estrutura de dados

• O algoritmo seguinte mostra uma primeira tentativa para o código de testar movimentos válidos

• Por questões de espaço mostraremos o código com o comando *for*, mas pode-se usar *while* e parar a execução tão logo se descubra que existe um movimento

Detalhamento das subrotinas

```
Function ExisteMovimento (var T: tabuleiro): boolean;
Var i,j: integer;
Begin
    ExisteMovimento:= false;
    For i:= 1 to max do
        For j:= 1 to max do
            If (T[i,j] = 0) or ExisteVizinhoIgual (T,i,j) then
                ExisteMovimento:= true;
End;
```

Detalhamento das subrotinas

- Como fazer para saber se existe vizinho igual?

- O teste é mais complicado para as bordas, por isto vamos criar uma borda no tabuleiro com valores nulos e evitar uma série de testes (*if*'s) que testam as bordas separadamente

Refinamento da estrutura de dados

- A nova matriz agora tem uma borda adicional, que só será utilizada pelo programador e não será exibida ao usuário

tabuleiro= array [**0**..MAX**+1**,**0**..MAX**+1**] of integer;

- Basta agora modificar a procedure IniciarJogo

Refinamento de IniciarJogo

```
Procedure IniciarJogo (var T: tabuleiro; var Pontos: integer);
Var i,j: integer;
Begin
    For i:= 0 to max+1 do
        For j:= 0 to max+1 do
            T[i,j]:= 0;
    For i:= 1 to 2 do
        InserirNovaPeca (T);
    Pontos:= 0;
End;
```

Detalhamento das subrotinas

- Desta forma o processo se torna muito simples, conforme mostrado no algoritmo seguinte

- Este algoritmo supõe que as coordenadas da posição em questão correspondem a uma posição válida da matriz e também que a borda foi corretamente inicializada

Detalhamento das subrotinas

```
Function ExisteVizinhoIgual (var T: tabuleiro; l,c: integer): boolean;
Begin
    ExisteVizinhoIgual:= false;
    If (T[l+1,c] = T[l,c]) or (T[l-1,c] = T[l,c]) or
       (T[l,c+1] = T[l,c]) or (T[l,c-1] = T[l,c]) then
            ExisteVizinhoIgual:= true;
End;
```

Detalhamento das subrotinas

- Em uma próxima aula poderemos estudar se existe uma técnica mais eficiente para este subproblema

- Por enquanto, só nos resta implementar uma última procedure:

MovimentarPecas (T,Direcao,Pontos)