**Departamento de Informática – DPI**

**Trabalho Prático 1 – INF 112**

**Alunos:**

Rodrigo Eduardo de Oliveira Bauer Chichorro – 92535

Matheus Ferreira Nunes – 92537

**Outubro – 2017**

**1. Introdução**

O trabalho consiste na criação de 2 códigos, nos quais cada um deles soluciona de forma diferente instâncias do jogo “fill-a-pix”.

**2. Resolução por Backtracking**

2.1. Introdução:

O primeiro código utiliza a estratégia de backtracking: o programa testará todas as combinações possíveis, preenchendo a matriz de saída posição a posição e, se o preenchimento realizado for inválido, o programa pula para a próxima combinação que não possui aquela combinação inválida.

Primeiramente o programa receberá como entrada o tamanho do tabuleiro e seus valores, assim como pedido na especificação. Em seguida, ele procurará a resposta (através da função “gerar\_combinacoes()”), e, ao encontrá-la, está será impressa, também no mesmo formato pedido na especificação. Logo após, os ponteiros serão deletados e o programa é encerrado.

2.2. Gerando as combinações:

No fill-a-pix, existem apenas duas opções para a resposta final: preto – representado pelo número 1 – e branco – representado pelo número 0 -. Assim, para um tabuleiro de 2x2, as combinações possíveis são:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Combinação | 1 1  1 1 | 1 1  1 0 | 1 1  0 1 | 1 1  0 0 | 1 0  1 1 | 1 0  1 0 | 1 0  0 1 | 1 0  0 0 | 0 1  1 1 | 0 1  1 0 | 0 1  0 1 | 0 1  0 0 | 0 0  1 1 | 0 0  1 0 | 0 0  0 1 | 0 0  0 0 |
| Representação  Horizontal | 1111 | 1110 | 1101 | 1100 | 1011 | 1010 | 1001 | 1000 | 0111 | 0110 | 0101 | 0100 | 0011 | 0010 | 0001 | 0000 |
| Índice | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Note que o número total que combinações é de 2m\*n, pois temos 2 opções (preto e branco, ou 1 e 0), *m* linhas e *n* colunas. Assim, para um tabuleiro 2x2, temos 22\*2 = 24 = 16 possíveis combinações. Note também que, se escrevermos a combinação de forma horizontal, as combinações serão números em binário, que decrescem de 2m\*n–1 até 0. Assim, temos uma forma confiável de gerar todas as combinações: começamos de 2m\*n–1 e vamos decrescendo até o 0.

Nó código, o número em binário foi representado como um arranjo de inteiros, em que o tamanho do arranjo é ‘int tam = m\*n’, que é o número de casas do tabuleiro e cada posição do arranjo guarda um algarismo do número binário. Assim, a primeira combinação será 2m\*n–1, que é preencher todas as posições do arranjo com 1. Isto é feito nas linhas 150-153 do código, dentro da função “gerar\_combinacoes()”:

int tam = m\*n;

int \*comb = new int[tam];

for(int i=0; i<tam; i++)

comb[i] = 1;

Em seguida, declaramos o arranjo “subtraendo”, que também representa um valor em binário. Ele possui o mesmo tamanho do arranjo “comb” e é inicializado com 0 em todas as suas posições. Seu uso será explicado mais tarde neste relatório.

int \*subtraendo = new int[tam];

for(int i=0; i<tam; i++)

subtraendo[i] = 0;

Em seguida temos um laço de *while*, que rodará enquanto o número binário representado por “comb” não for 0. Para isso, chamamos a função “binario\_igual\_a\_0()”, que retorna a soma dos algarismos de “comb”. Como os algarismos de “comb” sempre serão 1 ou 0, a soma de seus algarismos sempre será maior que 0, a não ser que todas as posições de “comb” sejam 0, que significa que chegamos na última combinação.

Assim, para cada iteração do while, testamos a combinação através da função “testa\_comb()”, que retorna um inteiro, a posição de “comb” na qual percebeu-se que aquela combinação era inválida. Caso o teste tenha tido sucesso, ou seja, a resposta daquela combinação é válida, retorna-se o valor -1. Este inteiro é atribuído para a variável inteira “jump”. “jump” possui este nome porque é responsável por pular combinações inválidas, ou seja, realizar o backtracking, que será explicado mais abaixo.

Caso nenhuma resposta tenha sido encontrada ao chegarmos na combinação 0, sairemos do laço *while*. Assim, “gerar\_combinacoes()” testará a combinação 0, pois, devido à condição de parada do *while*, esta não é testada no laço.

2.3. Testando as combinações:

Vamos explicar a implementação da função “testa\_comb()”, que é chamada na função “gerar\_combinacoes()”. Como sugere o nome, o que “testa\_comb()” faz é testar se uma combinação específica é resposta válida do problema.

Seus argumentos são: a matriz de entrada “a”; a matriz de saída “b”; as dimensões do tabuleiro “m” e “n”; a combinação que será testada “comb”; e o tamanho de “comb”, que é “tam”.

Assim, a matriz “b” é percorrida, e em cada iteração uma posição de “b” é atribuídas com o respectivo valor de “comb”. Para cada valor atribuído, a função verifica se a combinação ainda é válida. Se a combinação ainda é válida, a função continua a passar os valores de “comb” para “b” e testá-los. Caso contrário, a “testa\_comb” chama a função “limpar”, para voltar “b” ao seu estado original e retorna a posição de “comb” na qual a combinação foi declarada inválida.

Para verificar se a combinação ainda é válida, a programa observa as casas do tabuleiro que são afetadas pelo valores recém-atribuído, e verifica se alguma delas invalida a resposta. Exemplo:

|  |  |  |
| --- | --- | --- |
| 3 |  | 2 |
|  | 5 |  |
|  | 1 |  |

Suponha que, em uma de suas iterações, “testa\_comb()” tenha pintado a casa na qual o número 5 está localizada de preto. Note que as casas em branco representam casas que foram pintadas de branco e casas cinza são casas que ainda não foram pintadas. Assim que o programa pintou a casa na qual o número 5 está localizada de preto, ele verifica todos os valores ao redor da casa. No caso, estes são o 3, o 2, o 5 e o 1. Enquanto a resposta ainda é válida para o 3, o 5 e o 1, ela é inválida para o 2. Assim, “testa\_comb()” chama a função “limpar()”, que irá “despintar” a matriz “b” - na representação, passar as casas em preto e em branco de volta para cinza - e retornar como resposta a posição da casa na qual o 5 se localiza.

Já o segundo código preenche a matriz “da forma que um humano preencheria”. Ele utiliza diversas estratégias para “pintar” um espaço com absoluta certeza, e percorre a matriz indefinidamente, procurando casos que lhe permitam “pintar” espaços, ate que todos os espaços estejam preenchidos.

**3**