Universidade Federal de Viçosa

Rodrigo E.O.B. Chichorro - 92535
Matheus Ferreira Nunes - 92537

Trabalho Prático 1

Viçosa, 26 de outubro de 2017.

Universidade Federal de Viçosa

Rodrigo E.O.B. Chichorro - 92535
Matheus Ferreira Nunes - 92537

Trabalho Prático 1

Trabalho prático apresentado junto ao curso de Ciência da Computação, na Universidade Federal de Viçosa, como requisito à disciplina de INF112 - Programação II.

Viçosa, 26 de outubro de 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 2^{m*n}, pois temos 2 opções (preto e branco, ou 1 e 0), *m* linhas e *n* colunas. Assim, para um tabuleiro 2x2, temos $2^{2*2} = 2^4 = 16$ possíveis combinações. Note também que, se escrevermos a combinação de forma horizontal, as combinações serão números em binário, que decrescem de $2^{m*n}–1$ até 0. Assim, temos uma forma confiável de gerar todas as combinações: começamos de $2^{m*n}–1$ 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 $2^{m*n}–1$, 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:

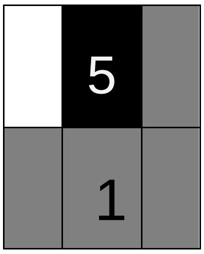
Vamos explicar a implementação da função “testa_comb()”, que é chamada na função “gerar_combinacoes()”. Como sugere o nome, o que “testa_comb()” faz é testar se uma combinação específica é resposta válida do problema.

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



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.

No entanto, caso a resposta seja válida, a função “limpar()” não é chamada, ou seja, a matriz “b” permanecerá com os mesmos valores, pronta para a saída da resposta.

2.4. Backtracking:

O uso de backtracking no programa ocorre quando, no meio de do teste de uma combinação, descobrimos que ela é uma resposta inválida. Como mencionado acima, “testa_comb()” retornará em qual posição descobrimos que a combinação é inválida, e este número é atribuído a “jump”.

Vamos utilizar o exemplo usado anteriormente:

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

Suponha que o programa esteja testando a combinação 1011. Ao atribuir o valor 0 na posição 1 (o arranjo começa no 0), o programa percebe que a combinação é inválida. Assim, o programa pode pular todas as combinações no formato 10XX, e próxima combinação a ser testada será a combinação 0111. Note que 1011 - 0111 = 11 - 7 = 4 = 0100, ou seja, pularemos 4 posições.

Note que 4 em binário é 0100, e que, ao representarmos este valor na forma de arranjo, temos que todas as suas posições valem 0, exceto a posição 1, justamente a posição na qual descobrimos que a combinação 1011 é inválida. Note também que, se 1011 - 0111 = 0100, então 1011 - 0100 = 0111, ou seja, subtraindo 0100 da combinação inválida, chegamos na próxima combinação que deve ser testada.

Assim, chegamos a um método para pular testes desnecessários: criamos uma arranjo com o mesmo tamanho de “comb”, chamado “subtraendo”.

Atribuímos o valor 0 a todas as posições de “subtraendo”, se uma resposta inválida for descoberta, atribuímos à posição “jump” de “subtraendo” o valor 1 (subtraendo[jump] = 1), e então, subtraímos “subtraendo” de “comb”, e a diferença entre eles será o novo valor de “comb”. Após a subtração, o valor de “subtraendo[jump]” volta a ser 0, pois ele não será mais necessário.

Essa estratégia faz com que o programa pule inúmeras combinações que não precisam ser testadas, obtendo, assim, em minutos, uma resposta que levaria anos caso não utilizássemos essa estratégia.

3. Resolução Alternativa

3.1 Introdução:

Já no segundo código, o programa preenche o campo da forma que um humano preencheria. Ele utiliza diversas estratégias, que são consequências das regras do jogo, para colorir um espaço com absoluta certeza, e percorre a matriz indefinidamente, procurando casos que lhe permitam “pintar” espaços, até que todos os espaços estejam preenchidos.

Devido ao fato de que essa solução não precisa “adivinhar” como completar o campo da forma certa, seu tempo de execução seria consideravelmente menor que a estratégia que usa apenas backtracking.

3.2 Nomenclaturas e definições:

No programa, as definições mais recorrentes são os arranjos de inteiros “a” e “b”, representados por um vetor de duas dimensões alocados dinamicamente. A matriz “a” serve como entrada, enquanto “b” é a saída, que será constantemente alterada durante a execução do código e impressa quando estiver completa.

Vale notar que, na matriz a, os valores inteiros diferentes de -1 representam a quantidade de blocos coloridos em volta do número em determinada posição, e -1 representa um bloco vazio. Além disso, em b, o valor -1 representa um bloco “indefinido”, ou seja, que não se sabe se é vazio ou preenchido, 0 indica que o bloco deve estar vazio, também referido como “X” pelo código, e 1 indica que o bloco está preenchido.

3.3 Execução das funções:

Assim que são geradas as matrizes de entrada e de saída e dadas as entradas, o programa chama a função “limpar”, que serve como uma espécie de construtor, definindo todos os blocos iniciais da matriz de saída b como vazios.

A partir daí, a função “fill()” passa a atuar. Na sua primeira parte, ela varre a matriz a apenas uma vez, a fim de achar casos triviais que geram algum resultado, como, por exemplo, no caso em que existe um bloco com o número 0 associado: o programa irá alterar todos os valores adjacentes ao bloco **na matriz b**, inclusive ele mesmo e os diagonais a ele, para 0, que representa um bloco vazio (com X).

O restante da função fill() roda de forma contínua até que não sobrem mais espaços vazios. Para isso, o programa irá sempre procurar dois casos principais:

1- Quando um bloco a[i][j] tem em b a contagem de blocos preenchidos igual ao seu número referente, é possível mudar todos os blocos não preenchidos ao redor de a para X.

No programa, isso equivale a dizer que se “num_black_blocks() == a[i][j]”, sendo a função anterior responsável por retornar um inteiro referente ao número de blocos preenchidos ao redor de a[i][j], então deve-se chamar a função “setX()”, que muda todos os blocos que não estejam preenchidos ao redor de a[i][j] para X (ou 0).

2- Quando um determinado bloco em a, chamado a[i][j], possui um número associado que é igual ao número de blocos preenchidos e vazios (com valor de -1) ao redor de si mesmo na matriz b, pode-se perceber que é necessário preencher os blocos vazios ao redor com valores 1 em b.

No código, isso significa que se “num_blocks_withoutX() == a[i][j]”, então a função setBlack deve ser chamada, para que todos os valores em b (diferentes de X) ao redor de a[i][j] sejam alterados para 1. Sendo “num_blocks_withoutX()” responsável por retornar a contagem de blocos vazios ou preenchidos (não se deve contar blocos com X),

3.4. Parte Complementar

Com apenas esses dois casos, é possível preencher boa parte da matriz, ou talvez até mesmo toda a matriz com 0’s ou 1’s para os fill-a-pix mais simples. Contudo, existem várias matrizes que não podem ser resolvidas com apenas esses dois casos, e existem diversos outros possíveis casos que podem ser usados, e adicioná-los todos ao código torna-se ineficiente.

Assim, o programa utiliza a lógica explicada anteriormente para preencher o maior número possível de casas. Em seguida, uma versão modificada do “gerar_combinacoes()” do primeiro programa é chamada, e o resto da matriz é completado utilizando backtracking.

4. Conclusões

Ao observar os resultados, embora não tenhamos realizado muitos testes, chegamos à conclusão que o segundo algoritmo é mais eficiente, pois ele elimina várias posições da matriz com as quais a parte de combinações não precisará se preocupar.

Por exemplo, com a entrada abaixo, o primeiro programa gasta vários minutos, enquanto o segundo gasta poucos segundos:

```
10 10
-1 -1 3 3 -1 -1 -1 -1 -1 -1
3 -1 -1 -1 -1 -1 0 -1 0 -1
-1 -1 3 4 -1 3 -1 -1 -1 -1
3 -1 4 -1 -1 -1 -1 3 -1 -1
2 3 -1 5 -1 4 4 -1 -1 4
-1 -1 5 4 6 6 -1 4 -1 4
-1 -1 -1 -1 -1 3 3 -1 -1 4
-1 3 -1 -1 5 6 5 -1 -1 4
```

-1 -1 -1 7 -1 -1 -1 7 -1 5
-1 4 -1 -1 6 -1 6 -1 5 -1