##MAC0121 - Algoritmo e Estruturas de Dados I##
Nome: Eduardo Hashimoto

nUSP: 6514136

Resta um

A lógica para resolução do problema foi semelhante a aplicada nos problemas apresentados em sala de aula.

Enquanto a configuração do tabuleiro não for a configuração final, o programa testa para cada uma das peças se é possível um movimento, se for ele faz, se não ele passa a próxima peça. Se a configuração do tabuleiro ficar estacionada em alguma configuração errada, algum movimento foi realizado erradamente e deve ser desfeito. A partir desse movimento desfeito seguimos o jogo para tentar achar a solução.
Se a pilha ficar vazia quer dizer que todo o *backtracking* foi feito e não há solução.

Abaixo o código comentado.

```
#include <stdio.h>
#include <stdlib.h>
#include "matriz.h"
#include "auxResta.h"
#include "pilha.h"

void resta(int **tab, int m, int n){

pilha *memoria;
pos atual;
int ok, mexeu;

int **tabInvertido;
tabInvertido = inverteTab(tab, m, n);

printf("configuração inicial\n\n");
imprimeMatriz(tab, m, n);

memoria = criaPilha(m*n);

atual.lin = atual.col = 0;
atual.mov = 1;

mexeu = 1;

while (!comparaMatrizes(tab, tabInvertido, m, n)){

        ok = 0;

        while(ePeca(tab, atual.lin, atual.col) && atual.mov <= 4 && ok == 0){

                if (podeComer(tab, atual.lin, atual.col, atual.mov, m, n)) {
                        ok = 1;
                }
                else
                        atual.mov++;
        }

        if (ok == 1){
```

```
                        empilha(memoria, atual);
                        come(tab, atual.lin, atual.col, atual.mov);
                        mexeu = 1;
                }

                else {

                        if (pilhaVazia(*memoria) && mexeu == 0 && atual.lin == m - 1
&& atual.col == n - 1){
                                printf("Impossible");
                                destroiPilha(memoria);
                                destroiMatriz(tab, m);
                                return;
                        }

                        if (mexeu == 0 && atual.lin == m - 1 && atual.col == n - 1) {
                                atual = desempilha(memoria);
                                volta(tab, atual.lin, atual.col, atual.mov);
                                atual.mov++;
                        }

                        else {
                                atual.mov = 1;
                                atual.col++;

                                if (atual.col == n){
                                        atual.col = 0;
                                        atual.lin++;
                                }

                                if(atual.lin == n){
                                        atual.lin = 0;
                                        mexeu = 0;
                                }
                        }
                }
        }

        printf("configuração final\n\n");
        imprimeMatriz(tab, m, n);
        printf("sequência de passos\n\n");
        imprimePilha(memoria);
        destroiPilha(memoria);
        destroiMatriz(tab, m);

        return;
        }
```

O código é bem direto e dispensa comentários, com exceção, talvez, da variável **mexeu**. Trata-se de um indicador de passagem que verifica se houve alguma mudança no tabuleiro ou seja, se o tabuleiro se encontra em estado estacionário.
Caso ela não existisse (de modo que eu usei, claro) o programa desempilharia assim que chegasse a ultima posição do tabuleiro, mesmo sem saber se aquela configuração é ou não estacionária.

Algumas funções e estruturas auxiliares foram usadas para completar a tarefa, a saber:

- estrutura pos, que armazena a posição (linha e coluna) e o movimento realizado

pela peça;
- estrutura de dados, pilha, com as funções criaPilha, realocaPilha, pilhaVazia, empilha, desempilha, destroiPilha e imprimePilha. Observar que se trata de uma pilha da estrutura pos, discutida acima;
- funções de matriz como criaMatriz, imprimeMatriz, iniciaMatriz, destroiMatriz, comparaMatriz.
- funções auxiliares relacionadas ao jogo, a saber:
-- ePeca, função se verifica se determinada posição do tabuleiro é uma peça, i.e, 1.
-- podeComer, função se verifica se para determinada posição e movimento, o movimento de troca de peças é válido;
-- come, função que altera a configuração do tabuleiro de acordo com a posição e o movimento admitidos pela função podeComer;
-- volta, função que altera a configuração do tabuleiro de acordo com a posição e o movimento resgatados da pilha.

Testes

O programa foi testado para as seguintes matrizes:

- 7 x 7 em cruz com apenas um buraco em [3][3] (configuração apresentada no EP);
- 6 x 6 quadrada com apenas um buraco em [2][3];
- 3 x 3 em 'T' com apenas um buraco em [2][0];

Para os quais foram obtidas as seguintes respostas, respectivamente:

1)
configuração inicial

```
0      0      1      1      1      0      0

0      0      1      1      1      0      0

1      1      1      1      1      1      1

1      1      1     -1      1      1      1

1      1      1      1      1      1      1

0      0      1      1      1      0      0

0      0      1      1      1      0      0
```

configuração final

```
 0      0     -1     -1     -1      0      0

 0      0     -1     -1     -1      0      0

-1     -1     -1     -1     -1     -1     -1

-1     -1     -1      1     -1     -1     -1

-1     -1     -1     -1     -1     -1     -1

 0      0     -1     -1     -1      0      0

 0      0     -1     -1     -1      0      0
```

sequência de passos

```
passo 0: 1:3 - 3:3,
passo 1: 2:1 - 2:3,
passo 2: 2:4 - 2:2,
passo 3: 2:6 - 2:4,
passo 4: 4:1 - 2:1,
passo 5: 4:3 - 4:1,
passo 6: 4:5 - 4:3,
passo 7: 4:6 - 2:6,
passo 8: 6:2 - 4:2,
passo 9: 6:4 - 6:2,
passo 10: 2:1 - 2:3,
passo 11: 2:3 - 2:5,
passo 12: 3:2 - 5:2,
passo 13: 4:0 - 4:2,
passo 14: 4:3 - 4:1,
passo 15: 6:2 - 4:2,
passo 16: 0:4 - 2:4,
passo 17: 2:0 - 4:0,
passo 18: 3:4 - 1:4,
passo 19: 5:4 - 5:2,
passo 20: 0:2 - 0:4,
passo 21: 0:4 - 2:4,
passo 22: 5:2 - 3:2,
passo 23: 4:0 - 4:2,
passo 24: 4:2 - 2:2,
passo 25: 1:2 - 3:2,
passo 26: 3:2 - 3:4,
passo 27: 3:4 - 1:4,
passo 28: 2:6 - 2:4,
passo 29: 1:4 - 3:4,
passo 30: 3:5 - 3:3.
[Finished in 81.8s]
```

2)

configuração inicial

```
1      1      1      1      1      1

1      1      1      1      1      1

1      1      1      -1     1      1

1      1      1      1      1      1

1      1      1      1      1      1

1      1      1      1      1      1
```

configuração final

```
-1     -1     -1     -1     -1     -1

-1     -1     -1     -1     -1     -1

-1     -1     -1     1      -1     -1

-1     -1     -1     -1     -1     -1
```

```
-1     -1     -1     -1     -1     -1

-1     -1     -1     -1     -1     -1

sequência de passos

passo 0: 0:3 - 2:3,
passo 1: 0:5 - 0:3,
passo 2: 1:1 - 1:3,
passo 3: 1:4 - 1:2,
passo 4: 2:5 - 0:5,
passo 5: 3:1 - 1:1,
passo 6: 3:3 - 3:1,
passo 7: 3:4 - 1:4,
passo 8: 4:1 - 2:1,
passo 9: 4:3 - 4:1,
passo 10: 4:5 - 4:3,
passo 11: 5:1 - 3:1,
passo 12: 5:3 - 5:1,
passo 13: 5:5 - 5:3,
passo 14: 0:2 - 0:4,
passo 15: 0:4 - 2:4,
passo 16: 1:2 - 3:2,
passo 17: 2:0 - 2:2,
passo 18: 2:2 - 4:2,
passo 19: 2:3 - 2:5,
passo 20: 3:5 - 1:5,
passo 21: 5:0 - 5:2,
passo 22: 5:3 - 5:1,
passo 23: 0:0 - 0:2,
passo 24: 0:5 - 2:5,
passo 25: 1:0 - 1:2,
passo 26: 3:0 - 5:0,
passo 27: 5:0 - 5:2,
passo 28: 5:2 - 3:2,
passo 29: 0:2 - 2:2,
passo 30: 3:1 - 3:3,
passo 31: 4:3 - 2:3,
passo 32: 2:2 - 2:4,
passo 33: 2:5 - 2:3.
[Finished in 17.3s]

3)

configuração inicial

1      0      0
1      1      1
-1     0      0

Impossível

[Finished in 2.3s]

Observações finais:

- logo de pronto imaginei que uma segunda variável de passagem fosse necessária, pois esse problema usa conjuntamente a questão do problema da nRainhas, que
```

considera a posição e do Passeio do Cavalo, que considera os movimentos realizados;
- pelo mesmo motivo acima, depois de alguma tentativa, imaginei que a estrutura *pos* deveria guardar também o movimento e não apenas as posições;
- em algum momento pensei que cada vez que eu empilhasse a estrutura *pos*, eu deveria voltar ao começo do tabuleiro. Isso no entanto, atrapalhou meu raciocínio a cerca do *backtracking* e me fez ficar com outra solução. Contudo, não verifiquei se é possível resolver o problema dessa maneira;
- inicialmente a comparação do laço mais externo era baseada nas quantidades de 1 e -1 do tabuleiro. Se a quantidade desses números fosse invertida, então a solução teria sido encontrada. No entanto, a especificação do problema diz "... deixar no final apenas uma peça exatamente na posição onde inicialmente havia o buraco...". Com aquela condição, o programa parava assim que encontrasse as quantidades invertidas sem se preocupar com a localização do buraco;
- o último problema resolvido para o programa finalmente funcionar foi a atualização da variável atual.mov. Se mesmo após o programa desempilhar atual.mov retorna a 1, o programa terá problemas, pois o *backtracking* não está sendo bem executado. Assim, criei uma condição: se houver desempilhamento o movimento a ser testado para a mesma célula desempilhada é o próximo (o mov "5" não afeterá o fluxo do programa e a próxima célula será testada).