Anderson Andrei da Silva, 8944025
Relatório EP2 – Resta Um

1.0 BACKTRACKING:

Backtracking é um tipo de algoritmo que representa um refinamento da busca por força bruta, em que múltiplas soluções podem ser eliminadas sem serem explicitamente examinadas. O termo foi cunhado pelo matemático estado-unidense D. H. Lehmer na década de 1950.

Uma busca inicial em um programa nessa linguagem segue o padrão busca em profundidade, ou seja, a árvore é percorrida sistematicamente de cima para baixo e da esquerda para direita. Quando essa pesquisa falha, ou é encontrado um nodo terminal da árvore, entra em funcionamento o mecanismo de backtracking. Esse procedimento faz com que o sistema retorne pelo mesmo caminho percorrido com a finalidade de encontrar soluções alternativas.

2.0 APLICAÇÃO LOCAL:

Aqui utilizaremos o backtracking para achar uma solução, caso exista, para o problema do resta um (modificado de acordo com o anunciado deste exercício programa).

Como dito à cima, o mesmo executa uma busca por força bruta, onde, ao encontrar um primeiro movimento válido repetirá o procedimento até que encontre uma solução ou chegue em um estado em que nada mais pode ser feito. Nesse caso ele retornará e irá desfazer tudo o que a última decisão alterou e então tentará verificar outras possibilidades diferentes. Ao fim, se ainda assim não encontrar nenhuma solução e tiver percorrido todos os “caminhos” possíveis, significa que realmente aquela instância do problema não tem solução.

Ou seja, aqui faremos testes sistemáticos e repetitivos em todas as posições do tabuleiro cuja se encaixe nas devidas condições. Ao encontrarmos um movimento válido o faremos, o que acarretará na mudança do tabuleiro e no início de uma nova busca por mais um movimento válido. E assim prosseguiremos até cairmos em um dos dois casos já descritos.

3.0 IMPLEMENTAÇÃO:

3.1 Entrada de dados:

A entrada será um tabuleiro, dado através de suas dimensões m × n (m é o número de linhas e n é o número de colunas) e de uma matriz m × n de {−1, 0, 1} com a seguinte interpretação: 0 representa uma posição não ocupada no tabuleiro, −1 representa uma posição com um buraco no tabuleiro e 1 representa uma posição ocupada com uma peça.

3.2 Saída de dados:

Caso nãoseja possível obter uma sequência de movimentos que resolve o problema seu programa deverá imprimir a palavra Impossivel. Caso contrário, você dever á imprimir uma sequência de movimentos que resolve o problema. Os movimentos deverão ser impressos um por linha no seguinte formato: x1:y1−x2:y2 , onde x1:y1 é a posição da matriz ocupada por uma peça que ficará vazia e x2:y2 é a posição do tabuleiro vazia que ficará ocupada por uma peça. Sabemos que neste movimento a posição intermediária deverá estar originariamente ocupada por uma peça e ficará vazia.

4. DINÂMICA DO ALGORITMO:

Como já foi dito, iremos procurar sistematicamente e repetitivamente peças que possam ser movidas, e então analisar uma possível sequência de movimentos a partir da primeira que for movida.

Estamos procurando pelos valores “1” no tabuleiro, que representam as peças, e ao encontrá-los verificamos sua vizinhança para verificar os movimentos possíveis para tal. A vizinhança é composta, sempre que possível, por elementos abaixo, à cima e aos lados da peça em foco. Ou seja, primeiro, dependemos da posição da peça no tabuleiro, pois se a mesma se encontrar nas margens (paredes ou limites da matriz que está sendo utilizada) do tabuleiro, seu número de vizinhos será menor. Por exemplo, seja o tabuleiro representado por tab(3x3); seja tab[0][0] = 1; temos então que essa posição só possuirá vizinhança em 2 direções: à direita (somando valores na segunda coordenada da posição da matriz) ou à baixo (somando valores na primeira coordenada).

Nessas vizinhanças estamos interessados em analisar, a partir do elemento em foco, os próximos 2 vizinhos em cada direção possível.

Para efeito de movimentação temos que: o movimento só é válido se o primeiro vizinho for 1 e o segundo for -1 (“espaços vazios”). Ainda utilizando o exemplo à cima, tab[0][0] teria um movimento válido se por exemplo tab[0][1] = 1 e tab[0][2] = -1.

Então, sendo no máximo quatro movimentos possíveis, os mesmos foram numerados de 0 a 3, respectivamente da seguinte maneira:

- 0 : movimento para à esquerda;
- 1 : movimento para baixo;
- 2 : movimento para à direita;
- 3 : movimento para cima.
-

Seguindo a lógica do programa, ao encontrarmos então uma peça, verificamos se essa poderá se mover para uma ou mais das 4 direções descritas. Entretanto, apenas 1 é analisada por vez, na ordem apresentada, e se caso for possível a peça é movimentada imediatamente e essa decisão (nova posição da peça movida e o movimento feito) é empilhada para futura possível alteração.

Após o movimento teremos então a alteração do tabuleiro da seguinte forma: As que eram peças virarão espaços vazios e o que era espaço vazio virará uma peça, ou seja, valores “1” serão substituídos por valores “-1” e o inverso também , sempre sendo feitos em triplas. Ou seja, serão alterados os valores da origem (peça) e do destino (espaço vazio) e por consequência, o que estiver no caminho, que no caso é outra peça.

A partir de um movimento possível encontrado e executado, o tabuleiro será percorrido desde o início para uma nova busca de possibilidades. Note que a cada movimento executado o tabuleiro é alterado, por isso a necessidade de se iniciar a busca do começo, pois alguma das alterações feita pode alterar o começo do tabuleiro, disponibilizando movimentos que ali não eram possíveis.

Caso cheguemos em um estado que nada mais pode ser feito, iniciaremos então o backtrack da seguinte maneira: desempilhamos a última decisão feita, desfazemos a mesma e então verificamos, se ao ser desfeito, o elemento em foco no momento está no fim do tabuleiro, ou seja, no limite da dimensão da coluna da matriz, se for esse o caso, iniciaremos a busca em uma nova linha e na primeira coluna da mesma. Caso contrário, apenas somaremos “1” no valor do índice coluna atual.

E então quando chegarmos no ponto em que a pilha de decisões se encontra vazia, significa que todas as decisões tomadas já foram desfeitas e não resta mais nenhum caminho para tentar percorrer o backtrack, assim sendo, o problema não tem solução.

5. FUNÇÕES

Para a implementação do programa, foram utilizadas as seguintes funções:

/* Struct que armazena os dados (linha,coluna e movimento feito) dos movimentos que serão empilhados*/

```
typedef struct {
        int l;
        int c;
        int mv;
} movimento;
```

/* Struct que define uma pilha */

```
typedef struct {
        int topo;
        movimento *v;
        int tam;
} pilha;
```

/* Recebe um inteiro n e devolve uma pilha de tamanhao n alocada dinamicamente. */

```
pilha *criaPilha (int n);
```

/* Recebe um ponteiro para um pilha e devolve 1 se ela estiver vazia ou 0 caso contrário.*/

```
int pilhaVazia(pilha *p);
```

/* Recebe um ponteiro para um pilha e devolve 1 se ela estiver cheia ou 0 caso contrário.*/

```
int pilhaCheia (pilha *p);
```

/* Recebe um ponteiro para um pilha, um ponteiro para movimento e um ponteiro para um inteiro tamanho (tam). Então desempilha o elemento do topo da pilha e altera diretamente os valores de movimento através do ponteiro recebido. */

```
void desempilha (pilha *p, movimento *mov, int *tam);
```

/* Recebe o ponteiro para um pilha, um ponteiro para movimento e um ponteiro para um inteiro tamanho (tam) e empilha o movimento na pilha. */

```
void empilha (pilha *p, movimento n, int *tam);
```

/* Recebe um ponteiro para pilha e devolve o elemento do topo.*/

```
int topoPilha(pilha *p);
```

/* Recebe um ponteiro para um pilha e libera o espaço que foi alocado para a mesma.*/

```
void destroiPilha(pilha *p);
```

/* Recebe um inteiro lin e devolve o enderaço de um vetor alocado dinamicamente de tamanho lin.*/

```
int *criaVetor(int lin);
```

/* Recebe um ponteiro de inteiro e libera o espaço alocado para ele.*/

```
void destroiVetor(int *v);
```

/* Recebe dois inteiros lin,col e devolve o endereço de uma matriz de dimensão mxn alocada dinamicamente. */

```
int **criaMatriz(int lin, int col);
```

/* Recebe o endereço de uma matriz, seu número de lin e colunas e zera todos os seus elementos. */

```
void zeraMatriz(int **tab, int lin, int col);
```

/* Recebe o endereço de uma matriz, seu número de lin e colunas e libera o espaço que foi alocado para tal. */

```
void destroiMatriz(int **tab, int lin, int col);
```

/* Recebe o endereço de uma matriz, seu número de lin e colunas e imprime a matriz */

```
void imprimeMatriz(int **tab, int lin, int col);
```

/* Recebe o endereço de duas matrizes, o número de linhas e colunas de ambas (que são de mesma dimensão) e troca (ou inverte) os valores da primeira da seguinte forma: 1 por -1 e -1 por 1, e armazena na seguunda */

```
void inverteMatriz (int **tab, int **tabInversa, int lin, int col);
```

/* Recebe o endereço de duas matrizes, o número de linhas e colunas de ambas (que são de mesma dimensão) compara todos os elementos de ambas verificando se todos são iguais. Se sim, devolve 1. Caso contrário, devolve 0. */

```
int comparaMatriz (int **tabA, int **tabB, int lin, int col);
```

/* Recebe um ponteiro para uma matriz tab, e inteiros p,q,mov,lin,col representando respectivamente a posição (p,q) do elemento, um movimento e a dimensão da matriz tab lin x col. Devolve se é possível fazer o movimento mov no tabuleiro tab dadas as condições dos paramentros recebidos. */

```
int podeMover(int **tab, int p, int q, int mov, int lin, int col);
```

/* Recebe um ponteiro para uma matriz tab, ponteiros para inteiros p,q, espaços e peças, e um inteiro mov representando respectivamente a posição (p,q) do elemento a ser movido, o número de espaços e peças no tabuleiro tab e o movimento em questão. O movimento mov é feito, e as alterações necessárias são feitas diretamentes no tabuleiro tab. */

```
void movePeca(int **tab,int *p, int *q, int mov, int *espacos, int *pecas);
```

/* Recebe um ponteiro para uma matriz tab, ponteiros para inteiros p,q, espaços e peças, e um inteiro mov representando respectivamente a posição (p,q) do elemento a ser movido, o número de espaços e peças no tabuleiro tab e o movimento em questão. O movimento mov é desfeito, e as alterações necessárias são feitas diretamentes no tabuleiro tab. */

```
void voltaPeca(int **tab,int *p, int *q, int mov, int *espacos, int *pecas);
```

/* Recebe um ponteiro de uma matriz de inteiros e dois inteiros representando sua dimensão e devolve 1 caso for possível inverter os 1s e -1s de tab, ou 0, caso contrário.*/

```
int restaN (int **tab, int lin, int col);
```

6.0 RESULTADOS:

Foram testadas alguns tabuleiros para efeito de testes no programa, onde todos os testados obtiveram resultados satisfatórios (os que eram possíveis foram feitos e os que não eram foi devolvido “Impossivel”). Alguns desses tabuleiros e o tempo de execução médio (pois variou as vezes quando testado) para eles são:

a) Tempo de execução: 0m0.002s
Tabuleiro:

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

b)Tempo de execução: 0m0.002s
Tabuleiro:

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

c) Tempo de execução: 0m0.037s
Tabuleiro:

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

d) Tempo de execução:0m0.002s
Tabuleiro:

1	1	1
1	1	1
1	1	1

7.0 REFERÊNCIAS

1.0 BACKTRACKING
https://pt.wikipedia.org/wiki/Backtracking#cite_note-1

3.1 Entrada de dados
3.2 Saída de dados
http://paca.ime.usp.br/pluginfile.php/100205/mod_resource/content/1/ep2.pdf