Capítulo 9

Recursividade com Retrocesso

" A análise de algoritmos é uma disciplina de engenharia. Um engenheiro civil, por exemplo, tem métodos e técnicas para prever o comportamento de uma ponte antes de a construir. Da mesma forma, um projectita de computação, deve ser capaz de prever o comportamento de um algoritmo antes de projectá-lo"

- Anômimo -

sumário:

9.1- Conceitos

A **Recursividade com retrocesso** (backtraking em inglês) é uma das técnicas de programação que utiliza algoritmos recursivos, com maior difusão, para resolver problemas computacionais, com incidência especial em problemas combinatórios.

Esta técnica de programação baseia-se em algoritmos que tentam encontrar várias soluções possíveis, denominadas por soluções candidatas. Essas soluções são construídas de forma incremental e podem ser abandonadas quando o algoritmo deteta que essa solução não leva a resolução do problema em geral.

Em termos mais precisos, o algoritmo inicia com uma solução hipotética. Mas quando descobre que essa solução não é a solução pretendida, porque está viola as condições do problema, o algoritmo retrocede, remove a solução hipotética e avança com uma outra solução hipotética.

São exemplos clássicos desta estratégia os seguintes problemas: o problema das oito rainhas no tabuleiro de xadrez, o problema dos movimentos do salto do cavalo no tabuleiro de xadrez e o problema de andar em labirintos.

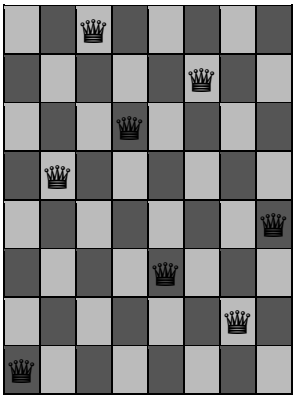
9.2- Exemplos

9.2.1- O Problema das 8 Rainhas

O problema das 8 rainhas, consiste em colocar essas rainhas num tabuleiro de xadrez, de tal modo que nenhuma rainha seja capaz de atacar outra.

Em outras palavras: escolher oito posições no tabuleiro de xadrez de forma que não se tenha duas rainhas a compartilharem a mesma linha, a mesma coluna ou a mesma diagonal.

O problema original possui 92 soluções distintas. Mas, se soluções simétricas ou obtidas por meio de permutações do tabuleiro forem unificadas, existem apenas 12 soluções, uma das possíveis soluções é mostrada a seguir:



A estratégia para resolver o problema consiste em colocar uma rainha por coluna, a começar na primeira coluna, e cada rainha é colocada na primeira linha disponível dessa coluna. Após a colocação de cada rainha, vamos marcar as casas que podem ser atacadas por ela, desta maneira estamos a excluir as casas que ficam indisponíveis para colocar outras rainhas. Quando chegarmos a uma situação que é impossível continuar, voltamos para trás, alterarmos a colocação da rainha anterior, de tal modo que podemos procurar uma disposição alternativa para as restantes rainhas.

Vamos mostrar em seguida essa estratégia com um exemplo descritivo, mas sugerimos ao leitor para utilizar um tabuleiro de xadrez para compreender melhor essa estratégia. Começamos por colocar a primeira rainha na primeira linha e na primeira coluna. Nesse caso, já não podemos colocar mais nenhuma rainha na primeira linha e na diagonal principal. Para colocar a segunda rainha na segunda coluna temos seis hipóteses. Vamos colocá-la na terceira linha. Sendo assim, restam quatro hipóteses para colocar a terceira rainha na terceira coluna. Vamos coloca-la na quinta linha. Agora para colocar a quarta rainha na quarta coluna, temos três hipóteses. Vamos colocá-la na segunda linha, agora temos duas hipóteses para colocar a quinta rainha. Vamos colocá-la na quarta linha. Nessa altura, toda a sexta coluna está ocupada, e não é possível colocar a sexta rainha. Agora, só nos resta abandonar esta solução e experimentar uma outra hipótese para colocar a quinta rainha. Colocamos essa rainha na oitava coluna. Só que esta hipótese também impede a colocação da sexta rainha na sexta coluna.

Uma vez que se esgotaram as hipóteses de colocação da quinta rainha, então, não nos resta outra alternativa que não seja voltar mais para trás, e experimentar uma outra alternativa para colocar quinta rainha.

Repare que esta estratégia é recursiva, em alguns casos, retrocede-se para configurações anteriores para explorar outras soluções parciais, até encontrar a solução parcial que leva a solução do problema.

Vamos em seguida, implementar esta estratégia. Antes, porém, vamos definir a estrutura de dados. O tabuleiro de xadrez pode ser visto como uma matriz quadrada de dimensão 8x8. Logo, vamos declarar as seguintes constantes simbólicas:

```
#define DIMTAB 8
```

e os possíveis valores que as casas do tabuleiro de xadrez podem assumir.

```
typedef enum { LIVRE = 0 , RAINHA = 1 , ATACADA = 2 } TCASA;
```

Com base nessa estrutura de dados, vamos desenvolver um procedimento para colocar 8 rainhas nesse tabuleiro.

```
void colocarRainhas ( TCASA tab[ ][DIMTAB], int nRainhas)
 {
   int lin, col, DAsc, DDesc;
   TCASA  tabAux;
    if (nRainhas == DIMTAB )  imprimirTabuleiro (tab);
    else {
            lin = 0;
            while ( lin < DIMTAB)
              {
                while (  tab[lin][nRainhas] != LIVRE && lin < DIMTAB ) lin++;
                 if (lin < DIMTAM)
                   {
                     copiarTab (tab , tabaux);
                      tab[lin][nRainhas] = RAINHA;
                      DAsc = lin-1, DDesc = lin + 1;
                      for (col = nRainhas+1; col < DIMTAB; col++ )
                        {
                          tab[lin][col] = ATACADA;
                          if (DAsc >= 0 ) tab[DAsc][col] = ATACADA;
                          if (DDesc >= 0) tab[DDesc][col] = ATACADA;
                          DAsc--;
                          DDes++;
                        }
                      colocarRainhas( tab , nRainhas+1);
                      copiarTab ( tabAux , tab);
                      tab[lin][nRainhas] = ATACADA;
                      lin++;
```

```
            }
        }
    }
}
```

O procedimento copiarTab(), tem por finalidade, copiar e recuperar o tabuleiro com as colocação das rainhas para evitar a redundância de código.

Em seguida, vamos implementar a função principal:

```
int main()
 {
   TCASA  tabuleiro [MAXTAB][MAXTAB];
    inicializarTabuleiro (tabuleiro);
    colocarRainhas (tabuleiro, 0);
     return 0;
 }
```

O procedimento inicializarTabuleiro(), consiste em preencher todos os campos da matriz com o valor 0 que significa que essas casas estão livres.

9.2.2- O Problema do Labirinto

Dado um labrinto representado por uma matriz de tamanho NxM, uma posição inicial $p_i = (x_i, y_i)$ e uma posição final $p_f = (x_f, y_f)$ tal que $p_i \neq p_f$. Determinar se existe um caminho entre p_i e p_f.

Podemos representar o labirinto como uma matriz M, tal que:

$$M(x,y) = \begin{cases} -2, se\ posição\ (x,y) representa\ uma\ parede \\ 1\ se\ a\ posição\ (x,y) não\ pertence\ ao\ caminho \\ i, para\ i > 0\ se\ a\ posição\ (x,y) pertence\ ao\ camniho \end{cases}$$

Neste caso, estamos a supor o labirinto é cercado por paredes, com excepsão de um elemento, uma casa designada por saída.

A figura a seguir mostra um labirinto com dimensão 8 x 8.

x	x	x	x	x	x	x	x
x	I						x
x	x		x				x
x			x	x	x		x
x		x	x				x
x		x				x	x
x				x			x
x	x	x	x	x	x	F	x

Onde:

X representa um obstáculo;
I o ponto inicial;
F o ponto final.

Uma possível estratégia consiste em adoptar os seguintes critérios: para a esquerda, para trás, para a direita e para frente. Com esses critérios obteríamos o seguinte percurso.

x	x	x	x	x	x	x	x
x	00	01					x
x	x	02	x				x
x	04	03	x	x	x		x
x	05	x	x				x
x	05	x	10	11	12	x	x
x	07	08	09	x	13	14	x
x	x	x	x	x	x	15	x

Uma outra estratégia consiste em adoptar os seguintes critérios: para a direita, para trás, para esquerda e para frente, obtendo desse modo o seguinte percurso.

x	x	x	x	x	x	x	x
x	00	01	02	03	04	05	x
x	x		x			06	x
x			x	x	x	07	x
x		x	x		08	08	x
x		x			10	x	x
x				x	11	12	x
x	x	x	x	x	x	13	x

É evidente que poderíamos utilizar outros critérios, mas deixamos que o leitor os defina e descreva os seus percursos.

Estamos em condições de desenvolver um procedimento em C, para determinar o percurso entre o ponto inicial e o ponto final. Antes, porém, vamos desenvolver um procedimento para carregar um labirinto numa matriz.

```
void criaLabirinto (int M[ ][MAX], int *n, int *m, int *Li, int *Ci, int *Lf, int
*Cf)
{
   int i, j, d;
   printf("\n Entre com as dimensoes do labirinto"); scanf("%d %d", n, m);
   printf("\n Entre com as coordenadas iniciais "); scanf("%d %d", Li, Ci);
   printf(\n Entre com as coordenadas finais "); scanf("%d %d", Lf, Cf);
   printf( " \n labirinto: 1 = parede ou obstaculo 0 = posicao livre ");
   for (i = 0 ; i < *n ; i++)
    for (j = 0 ; j < *m ; j++)
     {
       scanf("%d", &d);
```

```
            if (d == 1) M[i][j] = -2;
            else M[i][j] = -1;
          }
      }
```

Agora, vamos desenvolver uma função recursiva que entre outros parâmetros recebe a matriz do labirinto, as coordenadas iniciais e finais e, efectua o percurso do ponto inicial ao ponto final.

```
int labirinto (int M[ ][MAX], int deltaL[ ], int deltaC[ ],int linIni, int colIni,
int linFim, int colFim )
{
   int lin, col, k, passos;
   if ((linIni == linFim) && (colIni == colFim)) return M[linIni][colFim];
   /* testa todos os movimentos a partir da posicao atual */
   for (k = 0; k < 4; k++)
     {
       lin = linIni + deltaL[k];
       col = colIni + deltaC[k];
       /* verifica se o movimento é valido e gera uma solucao */
       if (M[lin][col] == -1)
         {
           M[lin][col] = M[linIni][colIni] + 1;
            passos = labirinto(M, deltaL, deltaC, lin, col, linFim, colFim);
            if (passos > 0) return passos;
          }
      }
   return 0;
}
```

Antes de efectuarmos a implementação da função principal, vamos formalizar os movimentos no labirinto. Suponhamos que os movimentos são baseados na seguinte estratégia: para à direita, para trás, para esquerda e para a frente. Então, a partir de qualquer ponto interior (x,y) é possível chegar aos pontos:

(x, y+1) ; (x, y+1); (x ,y-1) ; (x-1, y)

Esses movimentos podem ser armazenados em dois vectores com quatro elementos do tipo inteiro. O vector DeltaX[4], denominado por deslocamento do eixo X (abcissas) e o vector DeltaY[8] denominado por deslocamento do eixo y (ordenadas).

```
int main()
{
   int M[MAX][MAX], resp, n, m, linIni, colIni, linFim, colFim;
   /* define os movimentos validos no labirinto */
   int deltaX[4] = { 0, +1, 0, -1};
   int deltaY[4] = {+1, 0, -1, 0};
   /* carregar o labirinto */
```

```
    criarLabirinto (M, &n, &m, &linIni, &linIni, &linFim, &colFim);
   /* tenta encontrar um caminho no labirinto */
    posicaoInicial (M, linIni - 1, colIni - 1 );
    resp = labirinto (M, deltaL, deltaC, linIni-1, colIni-1, linFim-1, colFim-1);
    if (resp == 0)
       printf("\n Nao existe solucao.\n");
    else
      {
        printf("\n Existe uma solucao em %d passos.\n", resposta);
        imprimeLabirinto(M, n, m);
      }
    return 0;
}
```

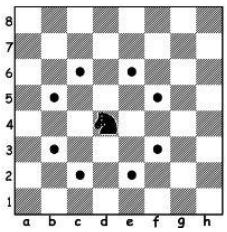
9.2.2- O Problema do Passeio do Cavalo

Dado um tabuleiro com NxN casas. A partir de uma posição inicial (x_0,y_0) determinar se existir, todos os movimentos do cavalo de tal forma que todas as casas do tabuleiro sejam visitadas uma e apenas uma vez.

Vamos em primeiro lugar estruturar a representação da informação. O tabuleiro de xadrez, pode ser representado por uma matriz T com dimensão 8x8 em que cada casa, possui a seguinte propriedade:

$$T(i,j) = \begin{cases} 0 \; se \; a \; posição \; (i,j) \; ainda \; não \; foi \; visitada \\ i \; se \; a \; posição \; do \; cavalo \; foi \; visitada \; no \; iésimo \; movimento \end{cases}$$

Os movimentos do cavalo são descritos pela seguinte figura:



O cavalo é uma peça do xadrez que possui um movimento peculiar. Ele movimenta-se em "L" e, pode pular sobre qualquer peça que esteja no seu caminho. Para além disso é a única peça que pode atacar uma Dama sem que esta possa atacar ao mesmo tempo.

Se um cavalo estiver posicionado por exemplo na casa d4 ele pode movimentar-se para as casas: f5, f3, e6, e2, c6, c2, b5 ou b3. Se houver alguma peça adversária em uma destas casas essa peça será captura pelo cavalo.

Então para qualquer posição (x,y) do tabuleiro, o cavalo pode mover-se para as seguintes casas:

(x+2,y+1), (x+1,y+2), (x-1,y+2), (x-2,y+1) , (x-2,y-1), (x-1,y-2), (x+1,y-2) e (x+2,y+1)

e esses movimentos podem ser armazenados em dois vectores com oito elementos do tipo inteiro. O vector DeltaL[8], denominado por deslocamento da linha e o vector DeltaC[8] denominado por deslocamento da coluna.

DeltaL	2	1	-1	-2	-2	-1	1	2
DeltaC	1	2	2	1	-1	-2	-2	-1

```
int passeio (int n, int x, int y, int nMov, int M[ ][MAX], int DL[ ], int DC[ ])
 {
   int k, nextL, nextCol;
   if (nMov == n*n) return 1;
   /* testa todos os movimentos a partir da posicao actual do cavalo (x,y) */
   for (k = 0; k < 8; k++)
    {
      nextLin = x + DL[k];
      nextCol = y + DC[k];
     /* verifica se o movimento é valido e gera uma solucao */
     if ((nextLinha >= 0) && (nextLinha < nMov) && (nextColuna >= 0) &&
         (nextColuna < nMov) && (M[nextLinha][nextColuna] == 0))
       {
         M[nextLinha][nextColuna] = nMov + 1;
         if (passeio(n, nextLinha, nextColuna, nMov + 1, M, nextLin,
            nextCol))  return 1;
         else  M[nextLinha][nextColuna] = 0; /* Liberta a posicao */
       }
    }
   return 0;
 }
```

Apresentamos em seguida a função principal que invoca a função que executa os movimentos do cavalo.

```
int main()
{
  int M[MAX][MAX], x, y, n, linha, coluna;
  int DeltaL[8] = {2, 1, -1, -2, -2, -1, 1, 2};
  int DeltaC[8] = {1, 2, 2, 1, -1, -2, -2, -1};
  printf("\n Entre com o valor de n: "); scanf("%d", &n);
  printf("\n Entre com a linha inicial do cavalo: "); scanf("%d", &linha);
  printf("\n Entre com a coluna inicial do cavalo: "); scanf("%d", &coluna);
  inicializaMatriz(n,M);
  posicaoInicial (M, linha - 1, coluna - 1);
  if ( passeio (n , linha-1 , coluna-1 , 1 , M , Dx , Y) == 0)
     printf("Nao existe solucao.\n");
  else
     imprimir (n , M);
  system("PAUSE");
  return 0;
}
```

Finalmente apresentamos uma função que irá mostrar na tela os n movimentos do cavalo. Esse procedimento recebe como parâmetros o número que movimentos solicitados pelo utilizador e a matriz que contêm os movimentos do cavalo

```
void imprimir (int n, int M[MAX][MAX])
 {
   int x, y;
   for ( x = 0 ; x < n ; x++)
    {
      for ( y = 0 ; y < n ; y++)
        printf(" %3d ", M[x][y]);
      printf("\n");
    }
 }
```

9.3- Recomendações Bibliográficas

Para o leitor aprofundar os seus conhecimentos sobre a estratégia de recursão com retrocesso, recomendamos os seguintes livros:

Levitin A.;- *Introduction to the Design and Analysis of Algorithms,* 3rd Edition, Pearson, 2011.

Sedgewick R. ;- *Algoritms in C*, Massachusetts, Addison-Wesley, Reading, Massachusetts,1990.

Rocha A. A.;- *Análise da Complexidade de Algoritmos*, Editora FCA, Portugal, 2014.

Ziviani N;- Projeto de Algoritmos com implementações em C e Pascal. Editora Thomson, 2a. Edição, 2004.

9.4- Exercícios Propostos

9.4.1-Desenvolva uma função para gerar todos os possíveis números de quatro dígitos utilizando um vetor, desde que cada posição do vector armazene apenas um dígito.

9.4.2- (Problema do Caixeiro viajante). Desenvolva uma função para encontrar o menor caminho que um caixeiro viajante leva a sair de uma cidade para outra, percorrendo todas as cidades da região uma e apenas uma vez.

9.4.3-(Problema de quatro cores).Desenvola uma programa com funções para colorir um mapa, de países reais ou imaginários, de forma que países com fronteira em comum tenham cores diferentes. Em 1852, Francis Guthrie conjecturou que 4 era esse número mínimo. Mas, não obstante a aparente simplicidade, só ao cabo de mais de cem anos, em 1976, se conseguiu provar que realmente a conjectura estava certa, obtendo-se o chamado Teorema das Quatro Cores.