AULA 25

Problema das n rainhas

PF 12

http://www.ime.usp.br/~pf/algoritmos/aulas/enum.html

http://en.wikipedia.org/wiki/Eight_queens_puzzle

Problema das n rainhas

Problema: Dado n determinar todas as maneiras de dispormos n rainhas em um tabuleiro "de xadrez" de dimensão $n \times n$ de maneira que duas a duas elas não se atacam.

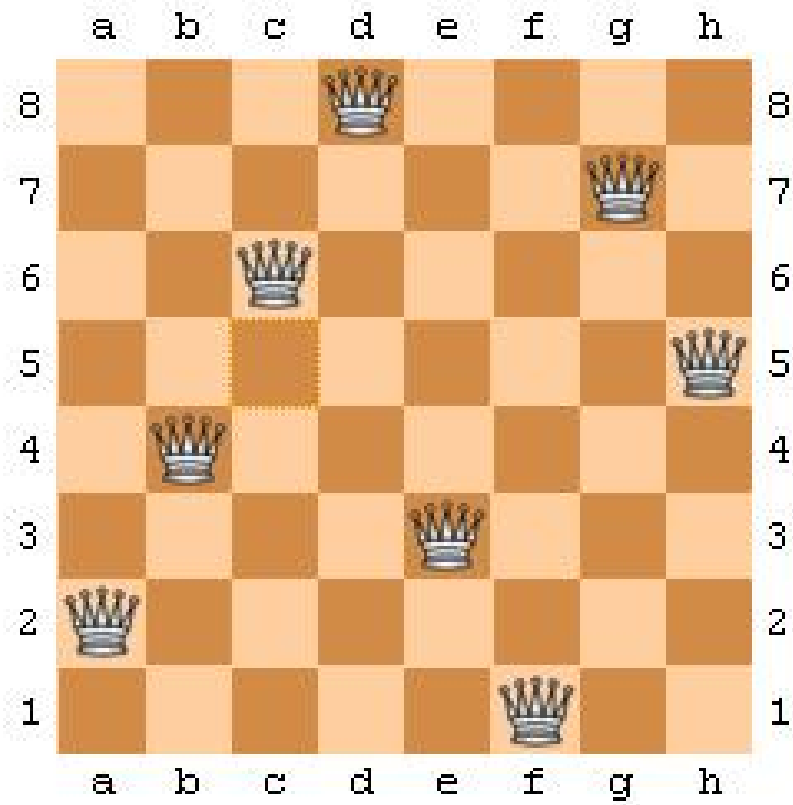
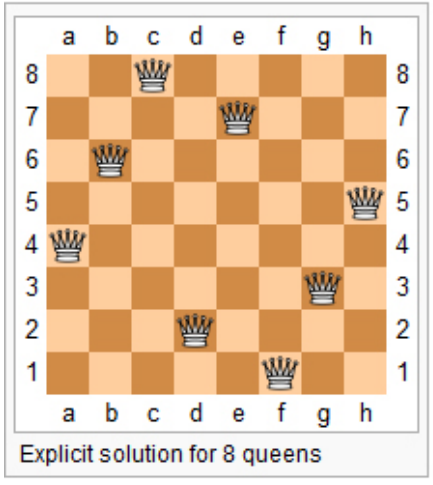
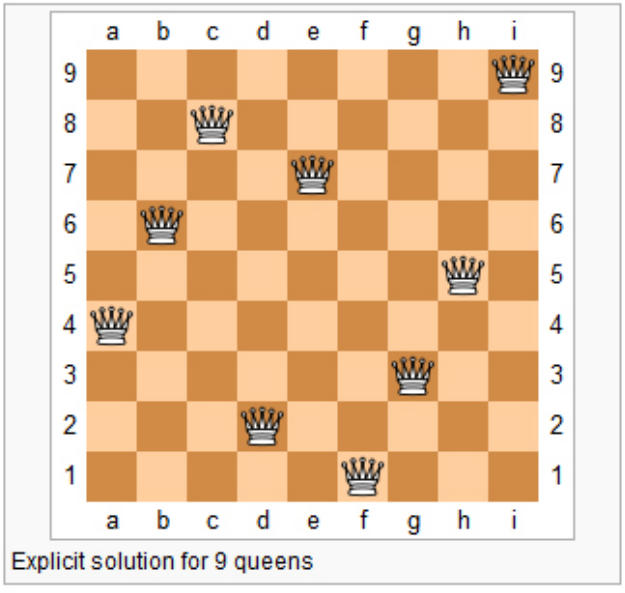


Imagem: http://software.intel.com/en-us/articles/multicore-enabling-the-n-queens-problem-using-cilk/

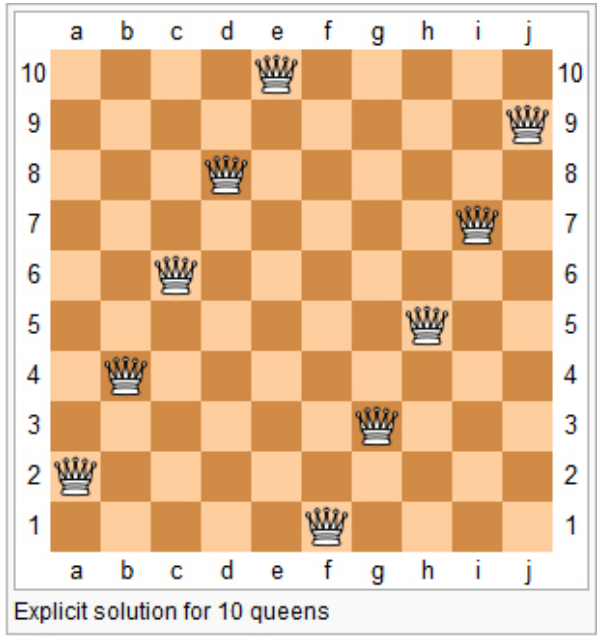
Soluções



Explicit solution for 8 queens



Explicit solution for 9 queens



Explicit solution for 10 queens

Imagem: http://www.levelxgames.com/2012/05/n-queens/

Problema das 8 rainhas

Existem $\binom{64}{8}$ maneiras diferentes de dispormos 8 peças em um tabuleiro de dimensão 8×8

$$\binom{64}{8} = 4426165368 \approx 4{,}4 \text{ bilhões}$$

Suponha que conseguimos verificar se uma configuração é válida em 10^{-3} segundos.
Para verificarmos todas as 44 bilhões gastaríamos

$$4400000 \text{ seg} \approx 73333 \text{ min} \approx 1222 \text{ horas} \approx 51 \text{ dias.}$$

Problema das 8 rainhas

Como cada linha pode conter apenas uma rainha, podemos supor que a rainha `i` será colocada na coluna `s[i]` da linha `i`.

Portanto as possíveis soluções para o problema são todas as sequências

$$s[1], s[2], \ldots, s[8]$$

sobre $1, 2, \ldots, 8$

Existem $8^8 = 16777216$ possibilidades. Para verificá-las gastaríamos

$$16777{,}216 \text{ seg} \approx 280 \text{ min} \approx 4{,}6 \text{ horas}$$

Problema das 8 rainhas

Existem outras restrições:

(i) para cada `i`, `j`, `k` $\neq$ `j`, `s`[`k`] $\neq$ `s`[`j`] (=duas rainhas não ocupam a mesma coluna); e

(ii) duas rainhas não podem ocupar uma mesma diagonal.

Existem $8! = 40320$ configurações que satisfazem (i).

Essa configurações podem ser verificadas em

$$\approx 40 \text{ seg.}$$

Função nRainhas

A função nRainhas a seguir imprime todas as configurações de n rainhas em uma tabuleiro $n \times n$ que duas a duas ela não se atacam.

A função mantém no início da cada iteração a seguinte relação invariante

(i0) $s[1..i-1]$ é uma solução parcial do problema

Cada iteração procura estender essa solução colocando uma rainha na linha i

Função nRainhas

A função utiliza as funções auxiliares

```
/* Imprime tabuleiro com rainhas em
      s[1..i] */
void
mostreTabuleiro(int n, int i, int *s);

/* Supoe que s[1..i-1] e solucao parcial,
 * verifica se s[1..i] e solucao parcial
 */
int solucaoParcial(int i, int *s);
```

Função nRainhas

```
void nRainhas (int n) {
   int i; /* linha atual */
   int j; /* coluna candidata */
   int nJogadas = 0; /* num.  da jogada */
   int nSolucoes = 0; /* num.  de sol.  */
   int *s = mallocSafe((n+1)*sizeof(int));
   /* s[i] = coluna da linha i em que
    * esta a rainha i, para i= 1,...,n.
    * Posicao s[0] nao sera usada.
    */
```

Função nRainhas

```
/* linha inicial e coluna inicial */
i = j = 1;
/* Encontra todas as solucoes.  */
while (i > 0) {
   /* s[1..i-1] e' solucao parcial */
   int achouPos = FALSE;
   while (j <= n && achouPos == FALSE) {
       s[i] = j;
       if (solucaoParcial(i,s) == TRUE)
           achouPos = TRUE;
       else j += 1;
   }
```

Função nRainhas

```
    if (j <= n) {/* AVANCA */
        i += 1;
        j = 1;
        if (i == n+1) {
            /* uma solucao foi encontrada */
            nSolucoes++;
            mostreTabuleiro(n,s);
            j = s[--i] + 1; /* volta */
        }
    } else { /* BACKTRACKING */
        j = s[--i]+1;
    }
}
```

Função nRainhas

```
    printf(stdout,"\n no.  jogadas = %d"
        "\n no.  solucoes = %d.\n\n",
        nJogadas, nSolucoes);
    free(s);
}
```

Rainhas em uma mesma diagonal

Se duas rainhas estão nas posições $(\mathtt{i}, \mathtt{j})$ e $(\mathtt{p}, \mathtt{q})$ então elas estão em uma mesma diagonal se

$$\mathtt{i} + \mathtt{j} == \mathtt{p} + \mathtt{q} \text{ ou } \mathtt{i} - \mathtt{j} == \mathtt{p} - \mathtt{q} \,.$$

Isto implica que duas rainhas estão em uma mesma diagonal se e somente se

$$\mathtt{i} - \mathtt{p} == \mathtt{q} - \mathtt{j} \text{ ou } \mathtt{i} - \mathtt{p} == \mathtt{j} - \mathtt{q} \,.$$

ou seja

$$|\mathtt{i} - \mathtt{p}| == |\mathtt{q} - \mathtt{j}|.$$

solucaoParcial

A função solucaoParcial recebe um vetor s[1 . . i] e supondo que s[1 . . i-1] é uma solução parcial decide se s[1 . . i] é uma solução parcial.

Para isto a função apenas verifica se a rainha colocada na posição (i, s[i]) está sendo ataca por alguma das rainhas colocadas nas posições

$$(1, s[1]), (2, s[2]), \ldots, (i-1, s[i-1]) \ .$$

solucaoParcial

```
int solucaoParcial(int i, int *s){
   int j = s[i];
   int k;
   for (k = 1; k < i; k++){
       int p = k;
       int q = s[k];
       if (q == j
           || i+j == p+q
           || i-j == p-q)
           return FALSE;
   }
   return TRUE;
}
```

Alguns números

nrainhas

n	jogadas	soluções	tempo
4	60	2	0.000s
8	15072	92	0.000s
10	348150	724	0.012s
12	10103868	14200	0.500s
14	377901398	365596	21.349s
15	2532748320	2279184	3m21s
16	?	14772512	18m48s

Backtracking

Backtracking (=tentativa e erro =busca exaustiva) é um método para encontrar uma ou todas as soluções de um problema.

A obtenção de uma solução pode ser vista como uma sequência de passos/decisões.

A cada momento temos uma solução parcial do problema. Assim, estamos em algum ponto de um caminho a procura de uma solução.

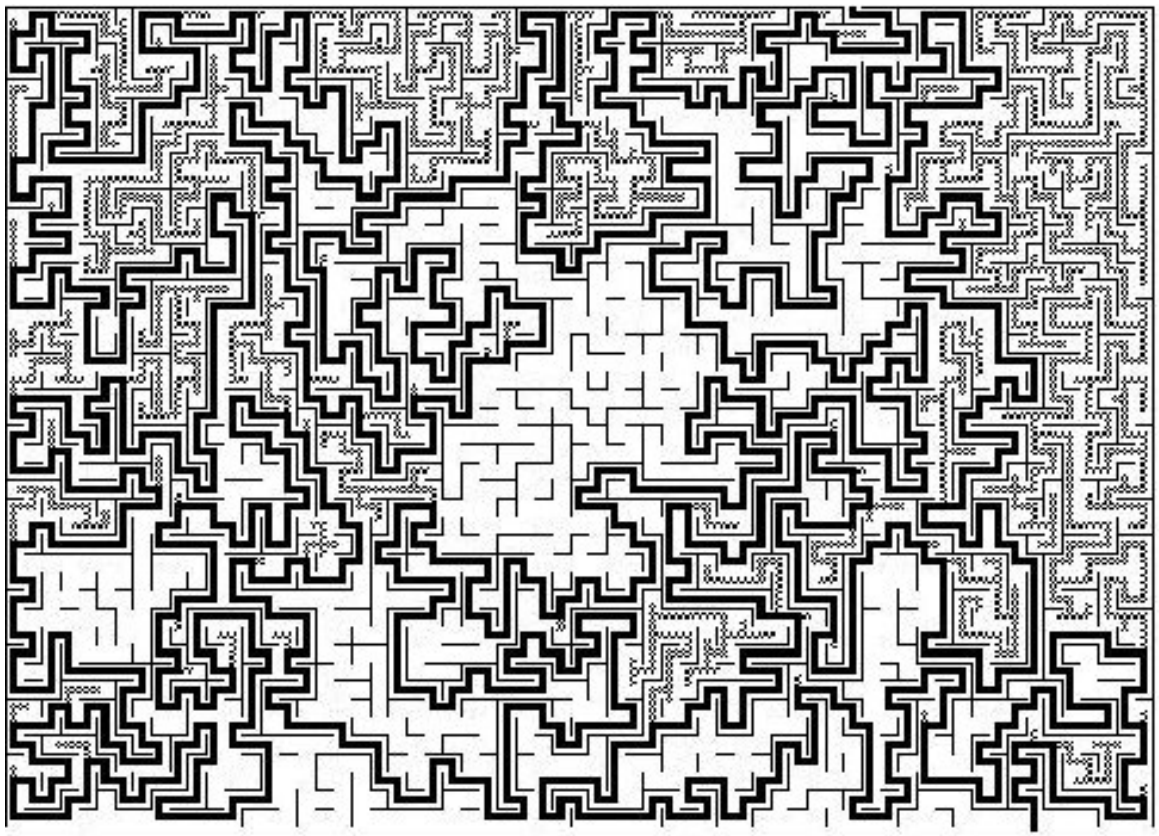
Backtracking

Cada iteração consiste em tentar estender essa solução parcial, ou seja, dar mais um passo que nos aproxime de uma solução.

Se não é possível estender a solução parcial, dar esse passo, **voltamos** no caminho e tomamos outra direção/decisão.

Backtracking

Para descrever *backtracking* frequentemente é usada a metáfora "procura pela saída de um labirinto".



Backtracking

A solução que vimos para o Problema das `n` rainhas é um exemplo clássico do emprego de *backtracking*.

No início de cada iteração da função `nRainhas` temos que `s[1 . . i-2]` é uma solução parcial:

> $(1, \texttt{s}[1]), \ldots, (\texttt{i}-1, \texttt{s}[\texttt{i}-1])$ são posições de rainhas que duas a duas elas não se atacam.

Árvore de estados

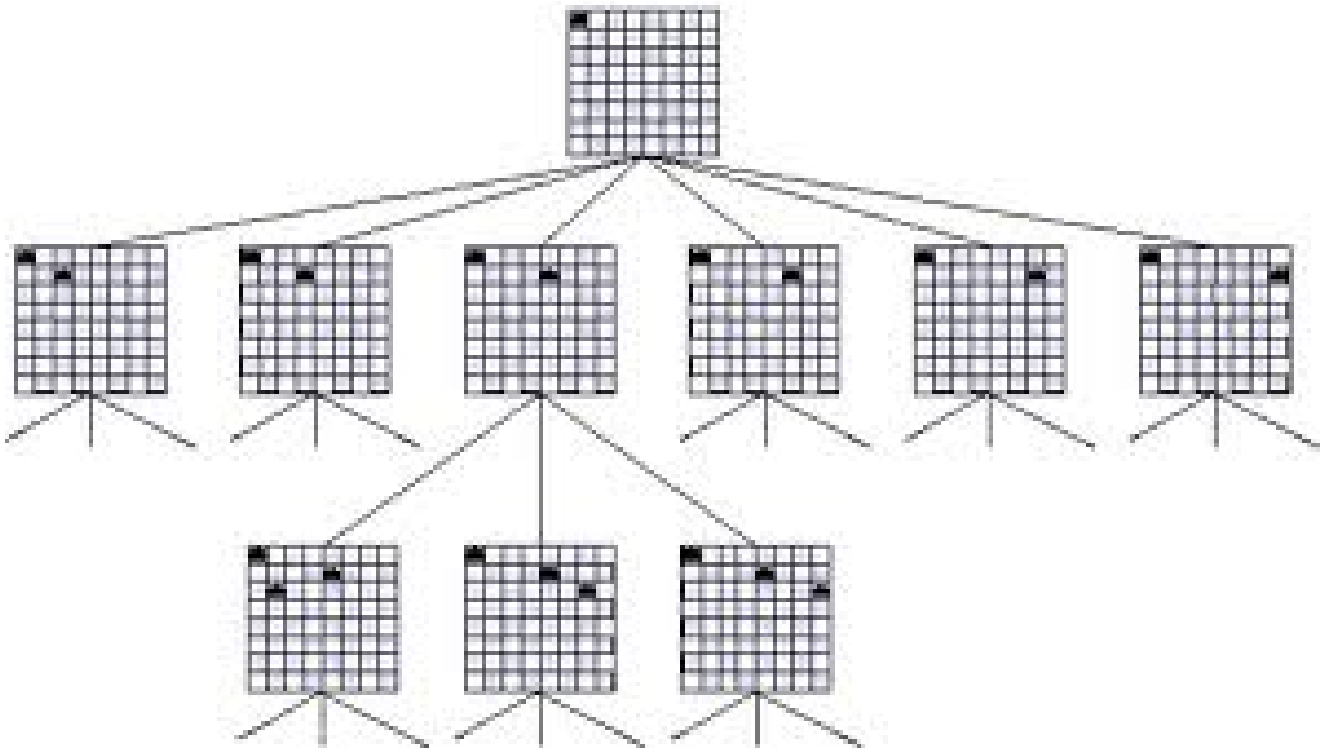


Imagem: http://cs.smith.edu/ thiebaut/transputer/chapter9/chap9-4.html

Árvore de estados

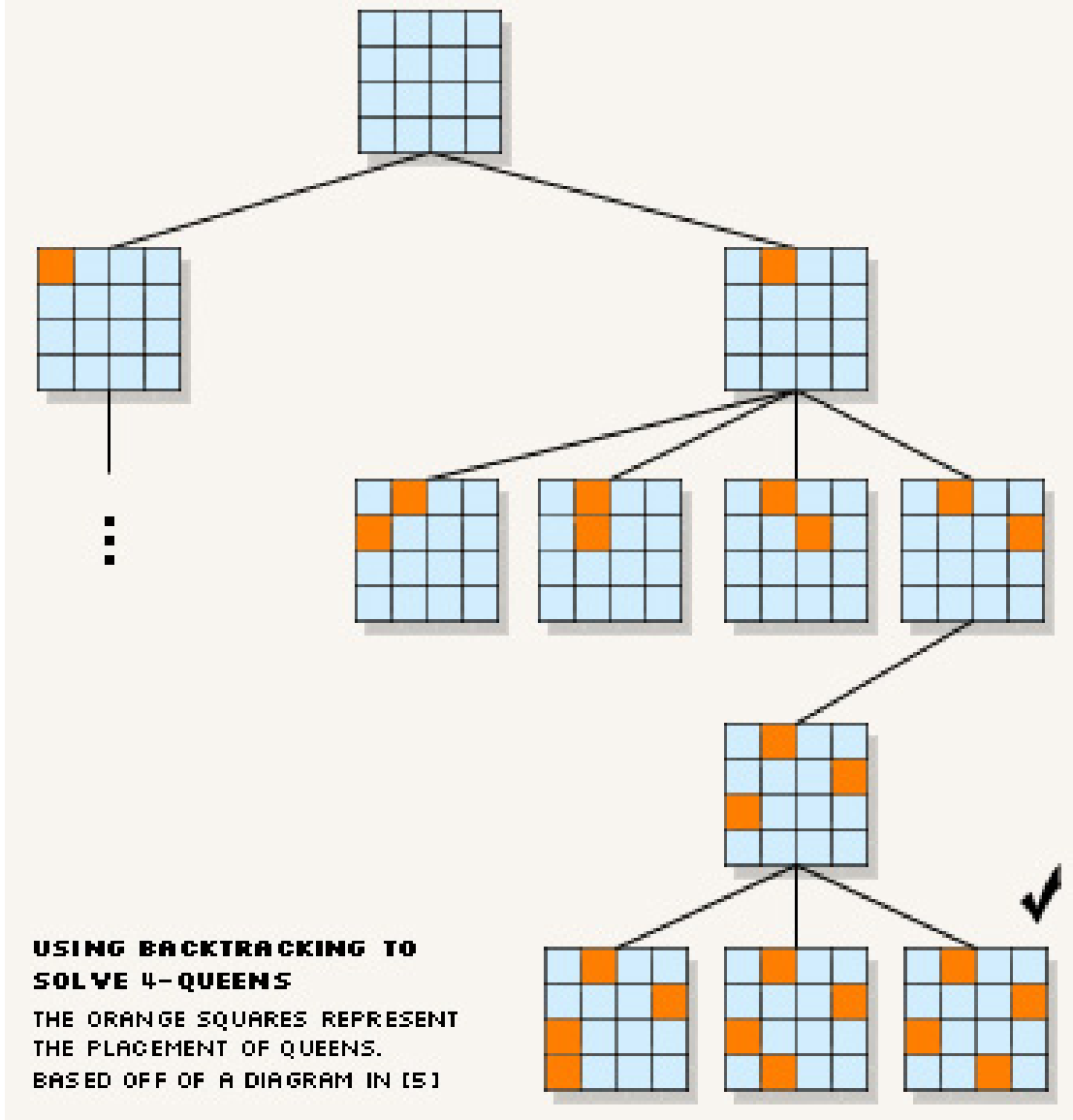


Imagem: http://4c.ucc.ie/web/outreach/tutorial.html

Função nRainhas (outra versão)

A função nRainhas a seguir imprime todas as configurações de n rainhas em uma tabuleiro $n \times n$ que duas a duas ela não se atacam.

A mantém no início da cada iteração a seguinte relação invariante

(i0) $s[1 \,..\, i-2]$ é uma solução parcial do problema

Cada iteração procura estender essa solução colocando uma rainha na linha i

Função nRainhas (outra versão)

A função utiliza as funções auxiliares

```
/* Imprime tabuleiro com rainhas em
      s[1..i] */
void
mostreTabuleiro(int n, int i, int *s);

/* Supoe que s[1..i-1] e solucao parcial,
 * verifica se s[1..i] e solucao parcial
 */
int solucaoParcial(int i, int *s);
```

Função nRainhas (outra versão)

```
void nRainhas (int n) {
   int testouTudo = FALSE;
   int i; /* linha atual */
   int j; /* coluna candidata */
   int nJogadas = 0; /* num.  da jogada */
   int nSolucoes = 0; /* num.  de sol.  */
   int *s = mallocSafe((n+1)*sizeof(int));
   /* s[i] = coluna em que esta a rainha i
    * da linha i, para i= 1,...,n.
    * Posicao s[0] nao sera usada.
    */
```

Função nRainhas (outra versão)

```
/* linha inicial e coluna inicial */
i = j = 1;
/* Encontra todas as solucoes.  */
while (testouTudo == FALSE) {
   /* s[1..i-2] eh solucao parcial */
   /* (i,j) e' onde pretendemos colocar
       uma rainha */

   /* CASO 1:  nLin == 0 */
   if (i == 0) {
       testouTudo = TRUE;
   }
```

Função nRainhas (outra versão)

```
/* CASO 2:  j == n+1 OU
    s[1..i-1]) nao e' solucao parcial */
else if (j == n+1 ||
        solucaoParcial(i-1,s)==FALSE){
    /* BACKTRACKING */
    /* voltamos para a linha anterior e
    * tentamos a proxima coluna
    */
    j = s[--i]+1; /* stackPop() */
}
```

Função nRainhas (outra versão)

```
/* Caso 3:  i == n+1 */
else if (i == n+1) {
    /* uma solucao foi encontrada */
    nSolucoes++;
    mostreTabuleiro(n,i-1,s);
    /* retira do tabuleiro a ultima
    * rainha colocada e volta
    */
    j = s[--i]+1; /* stackPop() */
}
```

Função nRainhas (outra versão)

```
    /* CASO 4:  j <= n &&
                s[1..i-1] e solucao parcial */
    else {
        /* AVANCA */
        s[i++] = j; /* stackPush() */
        j = 1;
        nJogadas++;
    }
}
```

Função nRainhas (outra versão)

```
    printf(stdout,"\n no.  jogadas = %d"
        "\n no.  solucoes = %d.\n\n",
        nJogadas, nSolucoes);
    free(s);
}
```