AULA 19

Problema das n rainhas

Fonte: http://www.bhmpics.com/

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.

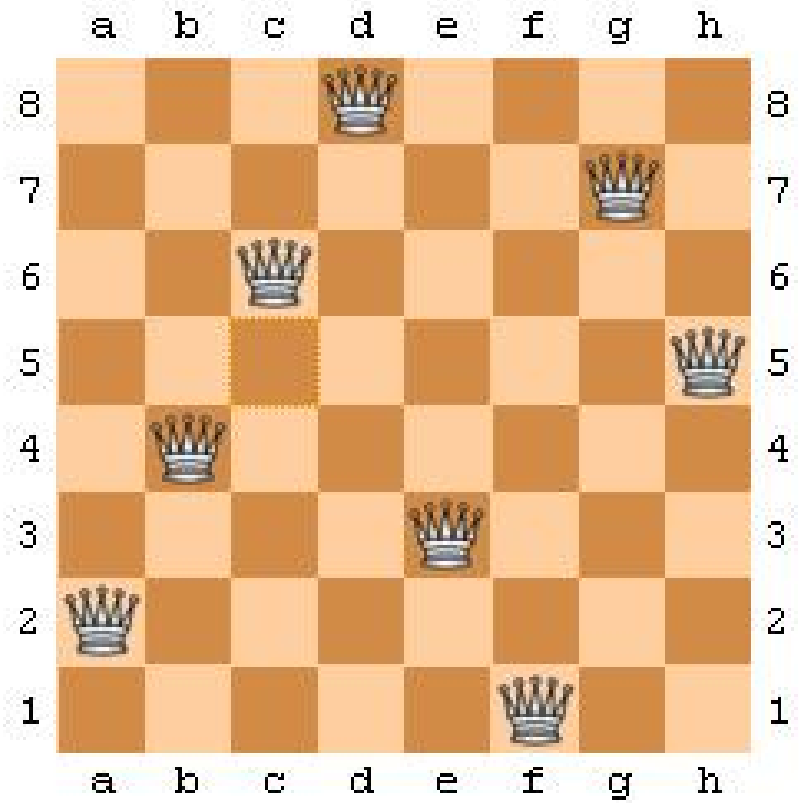
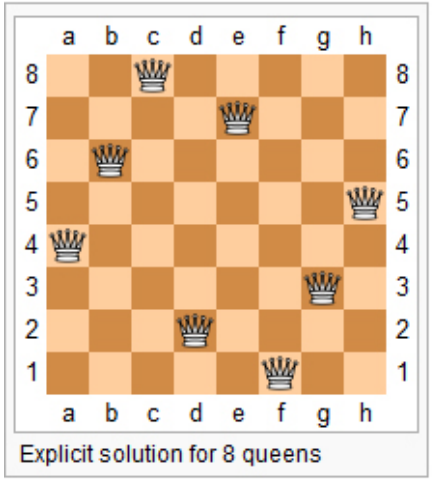
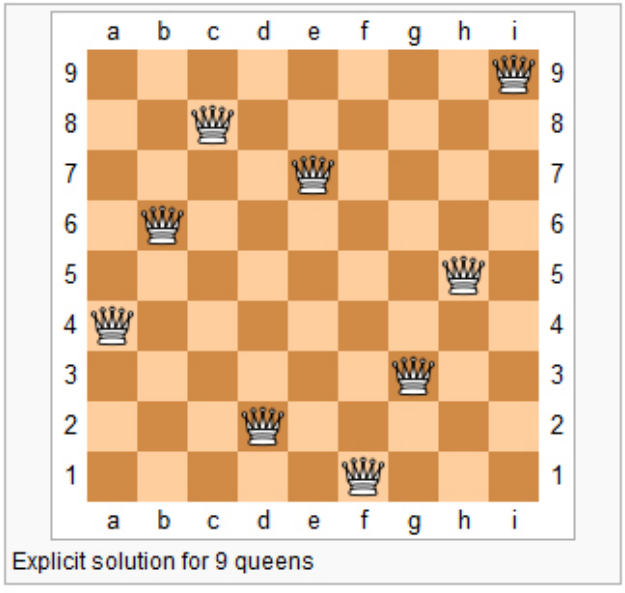


Imagem: http://software.intel.com/en-us/articles/multicore-enabling-the-n-queens-problem-using-cilk/

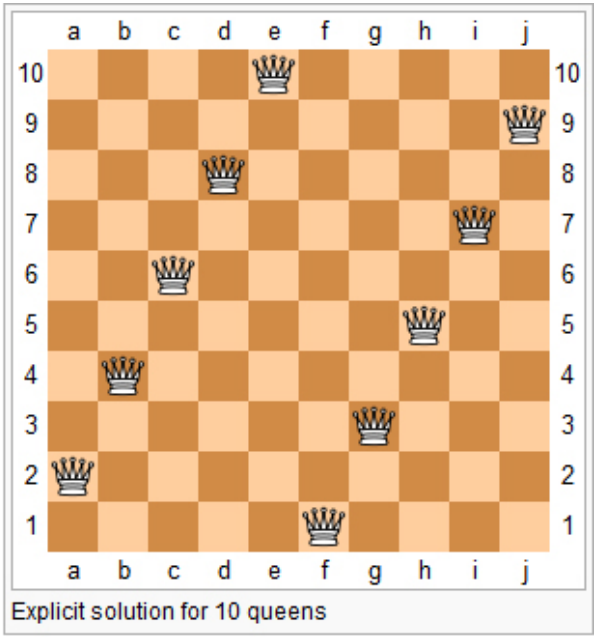
Soluções



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.

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

Problema das 8 rainhas

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

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

Suponha que conseguimos verificar se uma configuração é válida em 10^{-3} segundos.

Problema das 8 rainhas

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

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

Suponha que conseguimos verificar se uma configuração é válida em 10^{-3} segundos.

Para verificarmos todas as 4,4 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`.

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

$$\texttt{s}[1], \texttt{s}[2], \ldots, \texttt{s}[8]$$

sobre $1, 2, \ldots, 8$.

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

$$\texttt{s}[1], \texttt{s}[2], \ldots, \texttt{s}[8]$$

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`, $\texttt{i} \neq \texttt{j}$, $\texttt{s}[\texttt{i}] \neq \texttt{s}[\texttt{j}]$
(= duas rainhas não ocupam a mesma coluna);

Problema das 8 rainhas

Existem outras restrições:

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

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

Problema das 8 rainhas

Existem outras restrições:

(i) para cada `i`, `j`, $\texttt{i} \neq \texttt{j}$, $\texttt{s}[\texttt{i}] \neq \texttt{s}[\texttt{j}]$
(= 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 um tabuleiro `n` $\times$ `n` que duas a duas não se atacam.

Função nRainhas

A função nRainhas a seguir imprime todas as configurações de n rainhas em um tabuleiro $n \times n$ que duas a duas 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.

Função nRainhas

A função nRainhas a seguir imprime todas as configurações de n rainhas em um tabuleiro $n \times n$ que duas a duas 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 *s);
```

Função nRainhas

A função utiliza as funções auxiliares

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

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) {
      s[i] = j;
      nJogadas += 1;
      if (solucaoParcial(i, s))
         achouPos = TRUE;
      else j += 1;
   }
```

Função nRainhas

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

Função nRainhas

```
    if (achouPos) { /* 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 [i][j] e [p][q] então elas estão em uma mesma diagonal se

Rainhas em uma mesma diagonal

Se duas rainhas estão nas posições [i][j] e [p][q] então elas estão em uma mesma diagonal se

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

Rainhas em uma mesma diagonal

Se duas rainhas estão nas posições [i][j] e [p][q] então elas estão em uma mesma diagonal se

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

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

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

Rainhas em uma mesma diagonal

Se duas rainhas estão nas posições $[i][j]$ e $[p][q]$ então elas estão em uma mesma diagonal se

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

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

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

ou seja

$$|i - p| == |q - j|.$$

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.

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 atacada 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                          /* mesma coluna */
         || i+j == p+q                    /* mesma /-diagonal */
         || i-j == p-q)                   /* mesma \-diagonal */
         return FALSE;
   }
   return TRUE;
}
```

Alguns números

`nrainhas`

n	jogadas	soluções	tempo
4	16	2	0.00s
8	2056	92	0.00s
10	35538	724	0.01s
12	856188	14200	0.27s
14	27358552	365596	9.14s
15	171129071	2279184	54.00s
16	1141190302	14772512	5m55s
17	8017021931	95815104	41m27s

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.

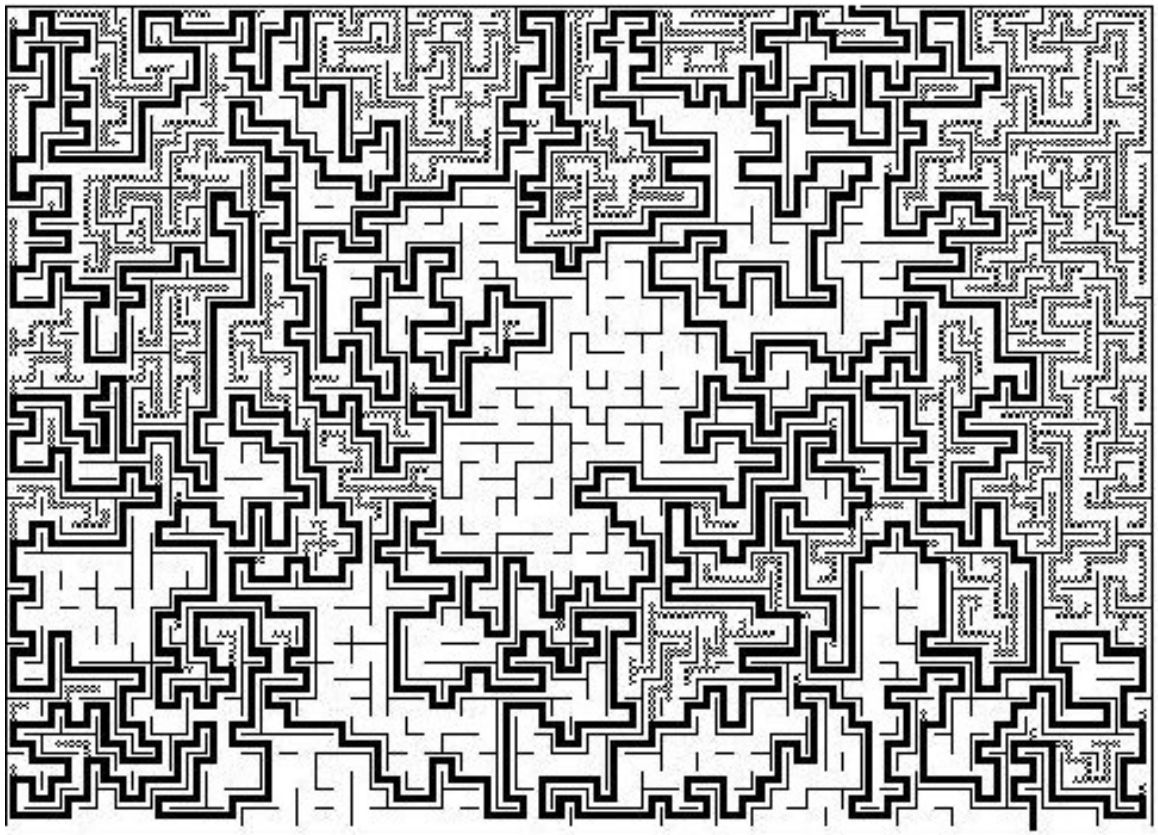
Backtracking

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, ou seja, dar nenhum 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*.

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`−1] 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 não se atacam.

Árvore de estados

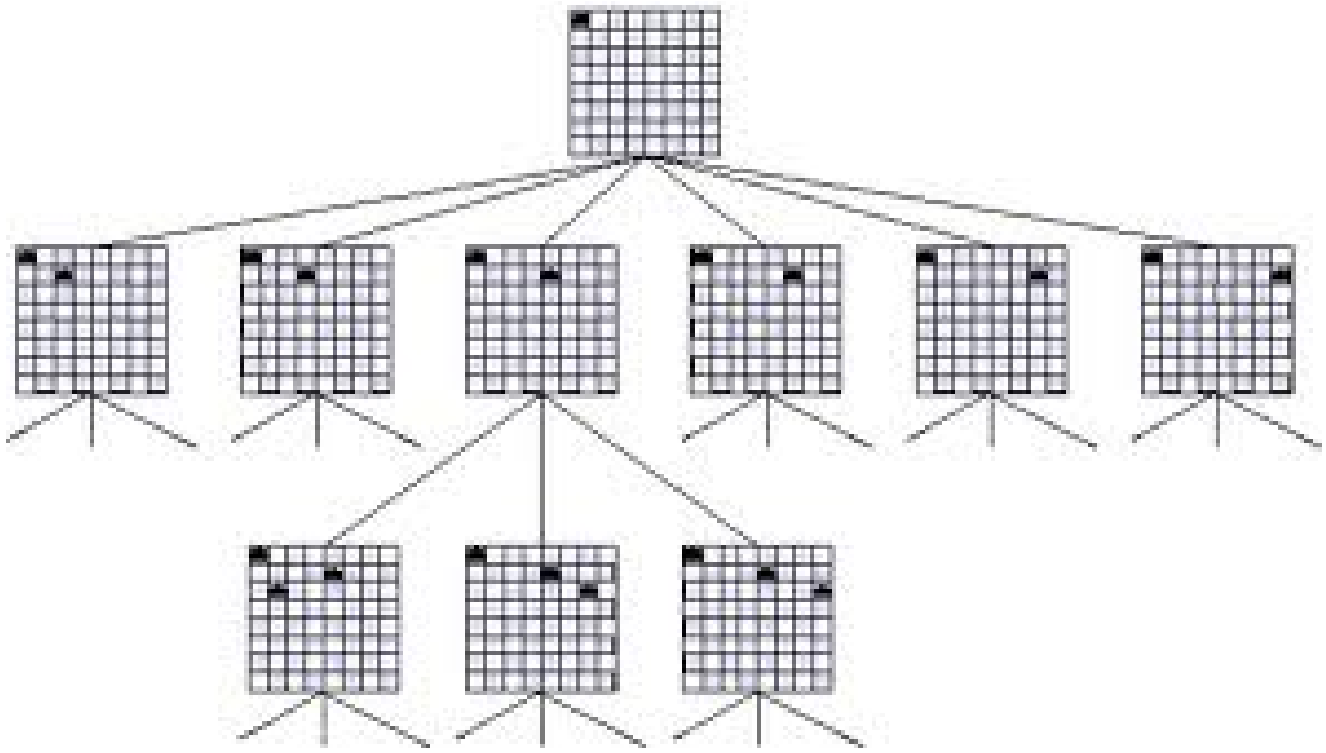


Imagem: http://cs.smith.edu/ thiebaut/transputer/chapter9/chap9-4.html

Árvore de estados

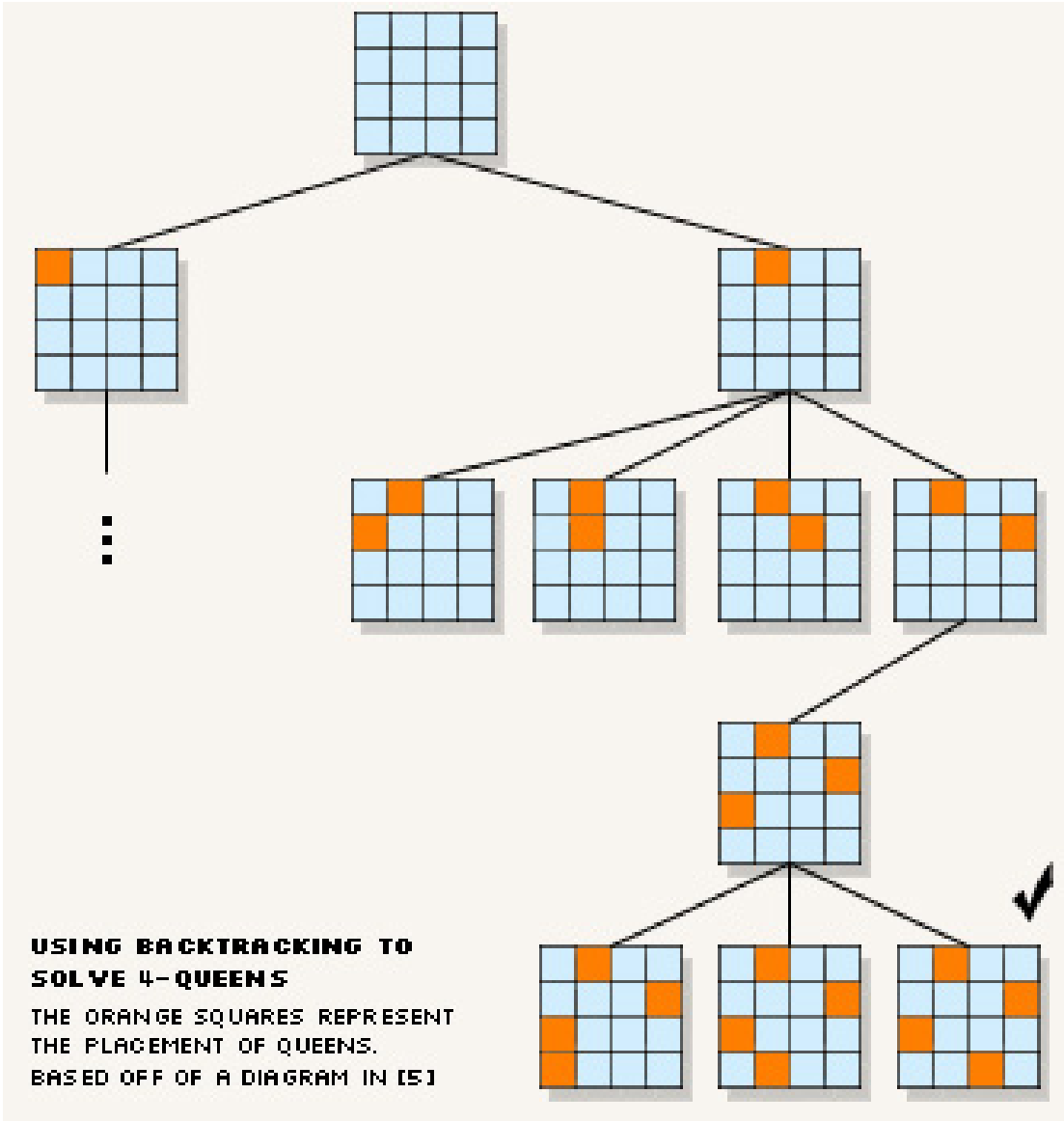


Imagem: http://4c.ucc.ie/web/outreach/tutorial.html

Função `nRainhas` (outra versão)

A função `nRainhas` a seguir imprime todas as configurações de `n` rainhas em um tabuleiro $\texttt{n} \times \texttt{n}$ que duas a duas não se atacam.

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

(i0) $\texttt{s}[1\,..\,\texttt{i}-1]$ 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 solucoes */
   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) {
   /* s[1..i-2] eh solucao parcial */
   /* [i][j] e' onde pretendemos colocar
      uma rainha */

   /* CASO 1:  i == 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);
}
```

Problema do passeio do cavalo

Fonte: `http://toonclips.com/design/5892`

"*Creating a program to find a knight's tour is a common problem given to computer science students*"

PF 12

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

http://en.wikipedia.org/wiki/Knight's_tour

Problema do passeio do cavalo

Problema: Num tabuleiro de xadrez $n \times n$, determinar se é possivel que um cavalo do jogo de xadrez parta da posição $(1,1)$ e complete um passeio por todas as n^2 posições.

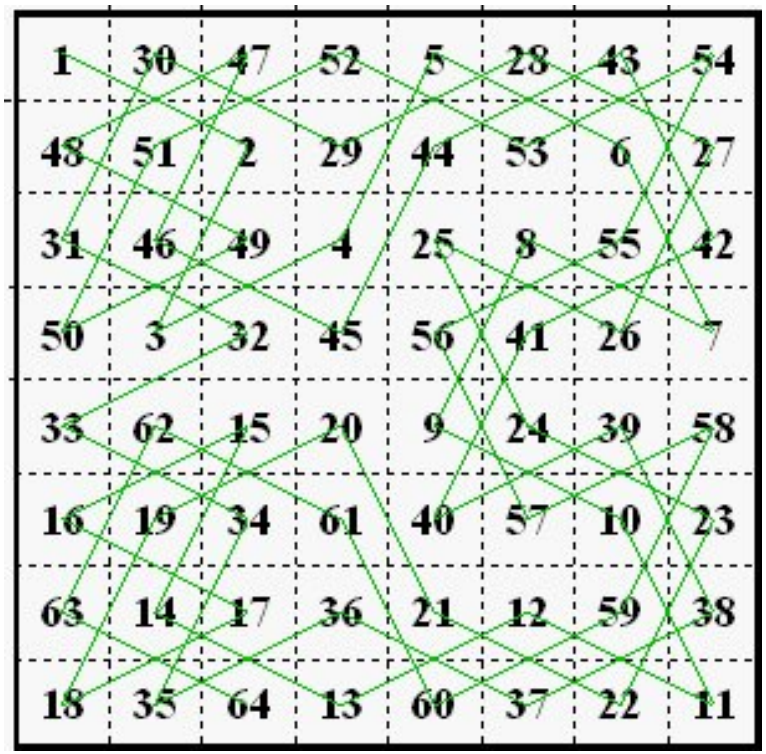


Imagem: http://www.magic-squares.net/knighttours.htm/

Soluções

1	1	10	19	4	25
2	18	5	2	9	14
3	11	20	15	24	3
4	6	17	22	13	8
5	21	12	7	16	23
	1	2	3	4	5

Soluções

1	1	14	11	28	7	4
2	12	27	2	5	10	29
3	15	20	13	8	3	6
4	26	33	24	19	30	9
5	21	16	35	32	23	18
6	34	25	22	17	36	31
	1	2	3	4	5	6

Soluções

1	1	18	27	6	11	16	13
2	28	7	2	17	14	5	10
3	19	26	29	8	3	12	15
4	40	45	20	25	30	9	4
5	37	34	39	44	21	24	31
6	46	41	36	33	48	43	22
7	35	38	47	42	23	32	49
	1	2	3	4	5	6	7

Movimentos do cavalo

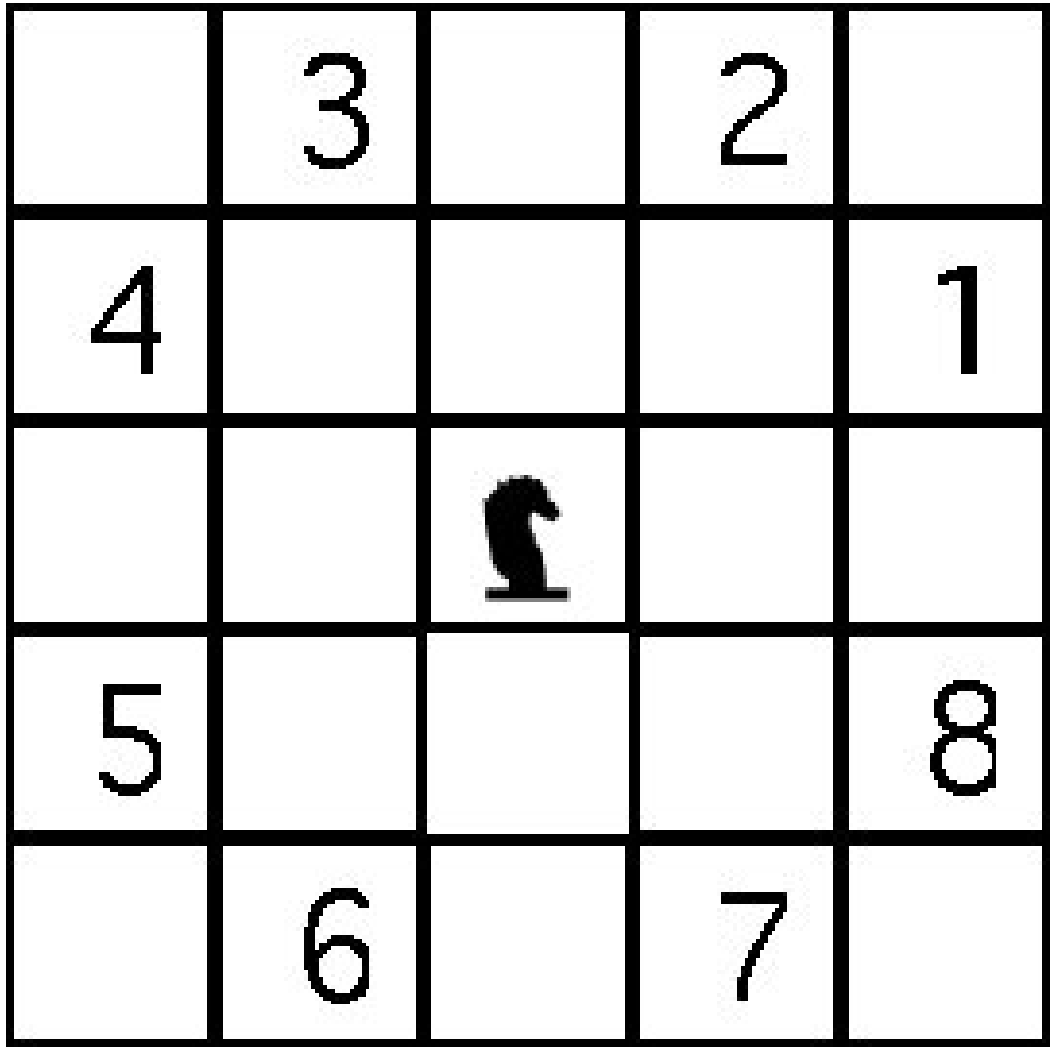


Imagem: http://www.mactech.com/articles/mactech/Vol.14/14.11/TheKnightsTour/index.html

Problema do passeio do cavalo

Cada movimento possível é de um dos tipos 1,. . . ,8 mostrados. De uma maneira grosseira existem 8^{63} sequência de movimentos em um tabuleiro 8×8.

$$8^{63} \approx 7.9 \times 10^{56}$$

Supondo que conseguimos verificar cada sequência em 10^{-6} segundos, o tempo para verificar todas as sequências seria

$$7.9 \times 10^{50} \text{ seg} \approx 1.310^{49} \text{ min} \approx 2.1 \times 10^{47} \text{ horas} \approx$$

Deixa para lá . . .

Algoritmo `passeioCavalo`

A função `passeioCavalo` a seguir imprime, caso exista, uma possível solução para o problema do passeio do cavalo em uma tabuleiro $\mathtt{n} \times \mathtt{n}$.

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

(i0) $\mathtt{s}[1\,..\,\mathtt{k}-1]$ são os movimentos de uma solução parcial do problema.

Cada iteração procura estender essa solução fazendo mais um movimento.

Algoritmo passeioCavalo

A função utiliza a função auxiliar e a struct

```
/* Imprime o tabuleiro com os movimentos. */
void mostreTabuleiro(int n, int **tab);

typedef struct {
   int i;
   int j;
} Movimento;
```

Algoritmo passeioCavalo

```
void passeioCavalo (int n) {
   int **tab;
   int i, j;                          /* posicao atual */
   int iProx, jProx;                  /* coluna candidata */
   int nMovimentos = 0;               /* num. de movimentos */
   int *s = mallocSafe((n*n+1)*sizeof(int));
   /* s[t] = movimento no passo t */
   int k;                             /* passo atual */
```

Algoritmo passeioCavalo

```
Movimento movimento[NMOV+1] = {
      {0,0},                          /* fica parado  */
      {-1,+2},                        /* movimento[1] */
      {-2,+1},                        /* movimento[2] */
      {-2,-1},
      {-1,-2},
      {+1,-2},
      {+2,-1},
      {+2,+1},
      {+1,+2}                         /* movimento[8] */
};
int mov;
```

Algoritmo passeioCavalo

```
/* linha 0 e coluna 0 do tabuleiro nao serao usadas */
tab = mallocSafe((n+1)*sizeof(int*));
for (i = 1; i <= n; i++) {
   tab[i] = mallocSafe(n+1, sizeof(int));
   for (j = 1; j <= n; j++) tab[i][j] = 0;
}

i = j = 1;                          /* posicao inicial */
k = 1;                                /* passo inicial */
mov = 1;                          /* movimento inicial */
tab[i][j] = 1;                    /* tabuleiro inicial */
```

Algoritmo passeioCavalo

```
while (0 < k && k < n*n) {
   int achouMov = FALSE;
   nMovimentos++;
   while (mov <= NMOV && !achouMov) {
      iProx = i + movimento[mov].i;
      jProx = j + movimento[mov].j;
      if ((0 < iProx && iProx <= n)
       && (0 < jProx && jProx <= n)
         && tab[iProx][jProx] == 0)
         achouMov = TRUE;
      else
         mov++;
   }
```

Algoritmo passeioCavalo

```
if (achouMov) { /* AVANCA */
   i = iProx;
   j = jProx;
   s[k] = mov;
   tab[i][j] = ++k;
   mov = 1;
}
```

Algoritmo passeioCavalo

```
    if (achouMov) { /* AVANCA */
       i = iProx;
       j = jProx;
       s[k] = mov;
       tab[i][j] = ++k;
       mov = 1;
    }  else { /* BACKTRACKING */
       tab[i][j] = 0;
       mov = s[--k];
       i -= movimento[mov].i;
       j -= movimento[mov].j;
       mov++;
    }
}
```

Algoritmo passeioCavalo

```
    if (k == n*n) {
       /* uma solucao foi encontrada */
       mostreTabuleiro(n, tab);
    }
    else printf("\n NAO TEM SOLUCAO\n");
    printf("Num. mov. = %d\n", nMovimentos);
    /* libera memoria alocada */
    free(s);
    for (i = 1; i <= n; i++) free(tab[i]);
    free(tab);
}
```