Continuação da aula passada

AULA 25

Busca de palavras em um texto

Dizemos que um vetor $p[1\,..\,m]$ **ocorre em** um vetor $t[1\,..\,n]$ se

$$p[1\,..\,m] = t[s+1\,..\,s+m]$$

para algum s em $[0\,..\,n{-}m]$.

Exemplo:

	1	2	3	4	5	6	7	8	9	10
t	x	c	b	a	b	b	c	b	a	x

	1	2	3	4
p	b	c	b	a

$p[1\,..\,4]$ ocorre em $t[1\,..\,10]$ com deslocamento 5.

Busca de palavras em um texto

Problema: Dados $\mathtt{p}[1\,..\,\mathtt{m}]$ e $\mathtt{t}[1\,..\,\mathtt{n}]$, encontrar o número de ocorrências de $\mathtt{p}$ em $\mathtt{t}$.

Exemplo: Para $\mathtt{n} = 10$, $\mathtt{m} = 4$, e

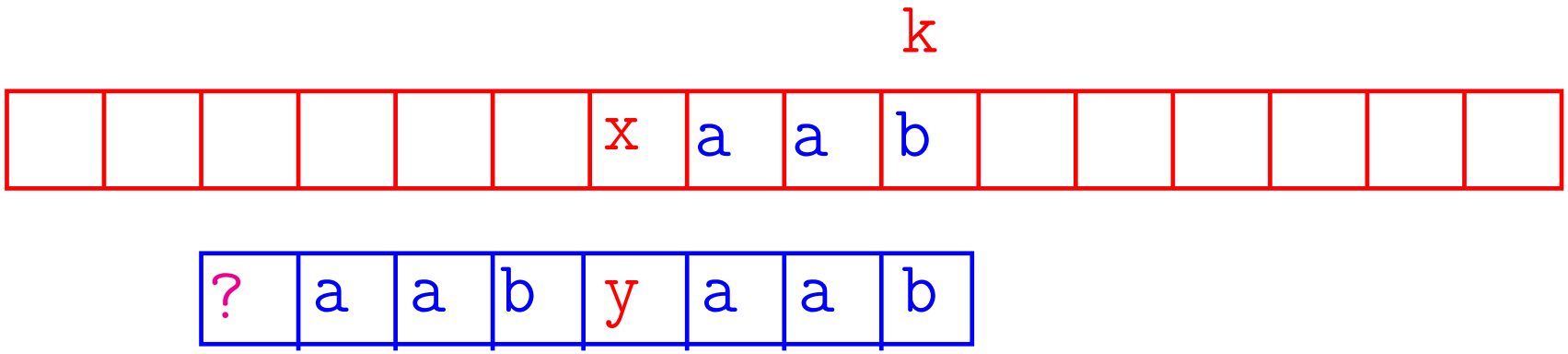
	1	2	3	4	5	6	7	8	9	10
t	b	b	a	b	a	b	a	c	b	a

	1	2	3	4
p	b	a	b	a

p ocorre 2 vezes em t.

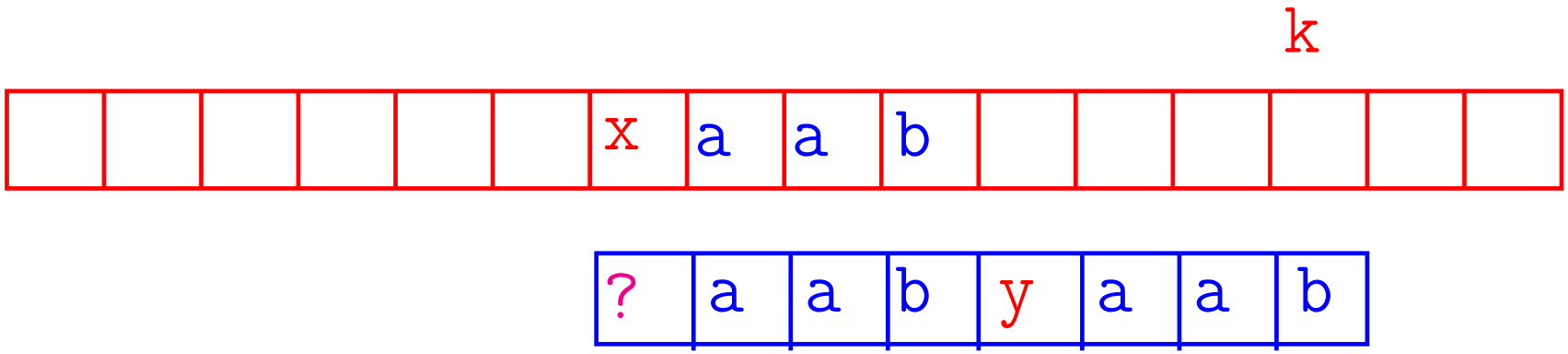
Segundo algoritmo de Boyer-Moore

O segundo algoritmo de R.S. Boyer e J.S. Moore (1977) é baseado na seguinte heurística.



Segundo algoritmo de Boyer-Moore

O segundo algoritmo de R.S. Boyer e J.S. Moore (1977) é baseado na seguinte heurística.



Segundo algoritmo de Boyer-Moore

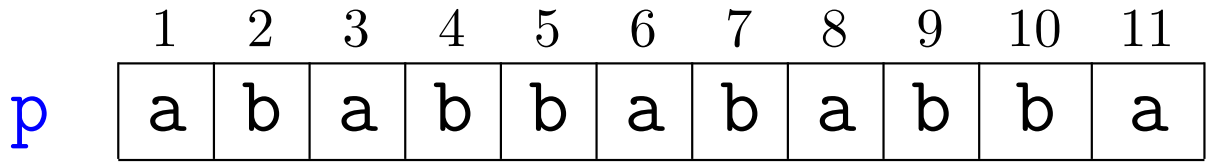
Ideia (*good-suffix heuristic*): para cada índice `i` calcular o maior `j` em 1 . . `m`-1 tal que

- `p`[1 . . `j`] é um sufixo de `p`[`i` . . `m`] ou
- `p`[`i` . . `m`] é um sufixo de `p`[1 . . `j`].

Para implementar essa ideia basta fazermos um pré-processamento que **só depende** de `p`.

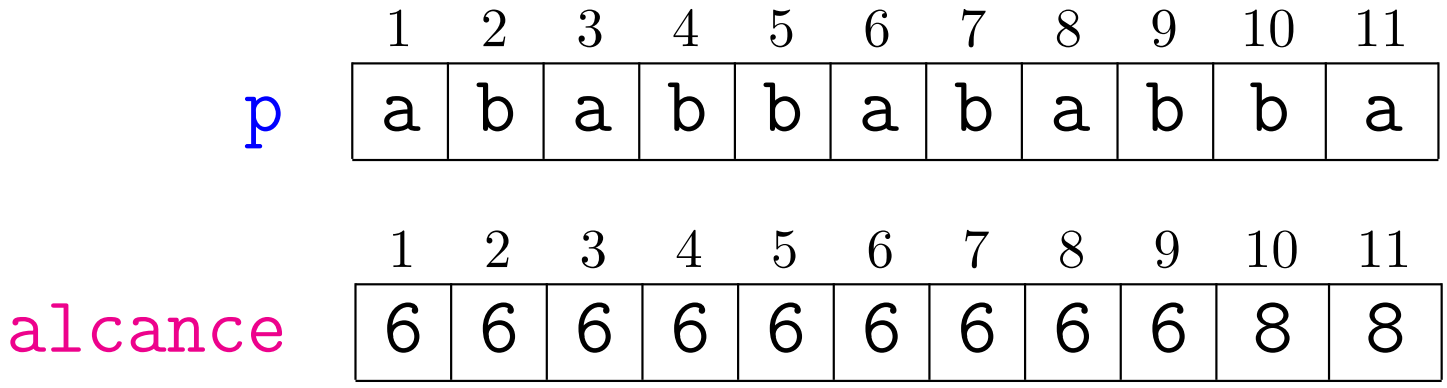
Segundo algoritmo de Boyer-Moore

Para implementar essa ideia, fazemos um pré-processamento de `p`, determinando para cada índice `i` o índice `alcance[i]` de um "sufixo bom".



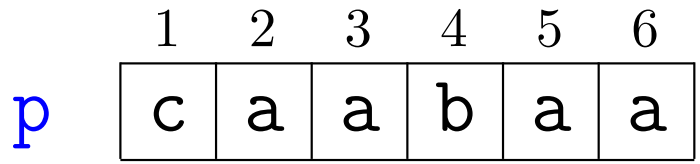
Segundo algoritmo de Boyer-Moore

Para implementar essa ideia, fazemos um pré-processamento de `p`, determinando para cada índice `i` o índice `alcance[i]` de um "sufixo bom".



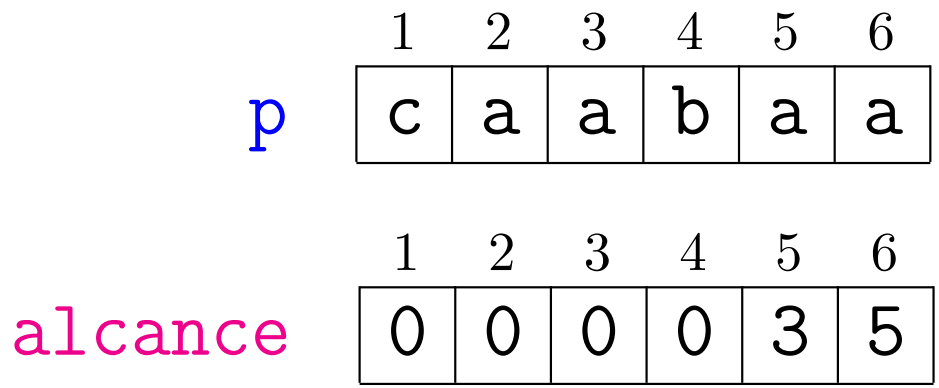
Segundo algoritmo de Boyer-Moore

Para implementar essa ideia, fazemos um pré-processamento de `p`, determinando para cada índice `i` o índice `alcance[i]` de um "sufixo bom".



Segundo algoritmo de Boyer-Moore

Para implementar essa ideia, fazemos um pré-processamento de `p`, determinando para cada índice `i` o índice `alcance[i]` de um "sufixo bom".



Segundo algoritmo de Boyer-Moore

Recebe vetores p[1 . . m] e t[1 . . n] de caracteres, com $m \geq 1$ e $n \geq 0$, e devolve o número de ocorrências de p em t.

```
int BoyerMoore2 (unsigned char p[], int m,
                 unsigned char t[], int n) {
   int *alcance;
   int i, r, k, ocorrs;
   /* pre-processamento da palavra p */
1   alcance = preProcessamento(p, m);
2   /* em branco */
```

Segundo algoritmo de Boyer-Moore

```
   /* busca da palavra p no texto t */
 3   ocorrs = 0; k = m;
 4   while (k <= n) {
 5       r = 0;
 6       while (r < m && p[m-r] == t[k-r])
 7           r += 1;
 8       if (r == m) ocorrs += 1;
 9       if (r == 0) k += 1;
10       else k += m - alcance[m-r+1];
     }
11   free(alcance);
12   return ocorrs;
   }
```

Pré-processamento

```
int *
preProcessamento(unsigned char p[], int m) {
    int i, r, j, *alcance;
    alcance = malloc((m+1)*sizeof(int));
1   for (i = m; i >= 1; i--) {
2       j = m-1; r = 0
3       while (m-r >= i && j-r >= 1)
4           if (p[m-r] == p[j-r]) r += 1;
5           else j -= 1, r = 0;
6       alcance[i] = j;
    }
7   return alcance;
}
```

Pior caso

p = a a a a a a a a a a a

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

Melhor caso

p = a a a a b

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
	?	?	?	c	b	?	?	?	c	b	?	?	?	c	b	?	?	?	c	b	?	?	?	t
1	a	a	a	a	b																			

Melhor caso

p = a a a a b

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
	?	?	?	c	b	?	?	?	c	b	?	?	?	c	b	?	?	?	c	b	?	?	?	t
1	a	a	a	a	b																			
2						a	a	a	a	b														

Melhor caso

p = a a a a b

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
	?	?	?	c	b	?	?	?	c	b	?	?	?	c	b	?	?	?	c	b	?	?	?	t
1	a	a	a	a	b																			
2						a	a	a	a	b														
3											a	a	a	a	b									

Melhor caso

p = a a a a b

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
	?	?	?	c	b	?	?	?	c	b	?	?	?	c	b	?	?	?	c	b	?	?	?	t
1	a	a	a	a	b																			
2						a	a	a	a	b														
3											a	a	a	a	b									
4																a	a	a	a	b				

Melhor caso

p = a a a a b

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
	?	?	?	c	b	?	?	?	c	b	?	?	?	c	b	?	?	?	c	b	?	?	?	t
1	a	a	a	a	b																			
2						a	a	a	a	b														
3											a	a	a	a	b									
4																a	a	a	a	b				
5																					a	a	a	...

Conclusões

O consumo de tempo da função `BoyerMoore2` no pior caso é $\mathrm{O}((\mathtt{n}-\mathtt{m}+1)\mathtt{m})$.

O consumo de tempo da função `BoyerMoore2` no melhor caso é $\mathrm{O}(\mathtt{n}/\mathtt{m})$.

Isto significa que no pior caso o consumo de tempo é essencialmente proporcional a $\mathtt{mn}$ e no melhor caso o algoritmo é **sublinear**.

Terceiro algoritmo de Boyer-Moore

O algoritmo de Boyer-Moore propriamente dito é uma fusão dos dois anteriores:

> *a cada passo, o algoritmo escolhe o maior dos deslocamentos ditados pelas tabelas* `ult` *e* `alcance`*.*

AULA 26

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.

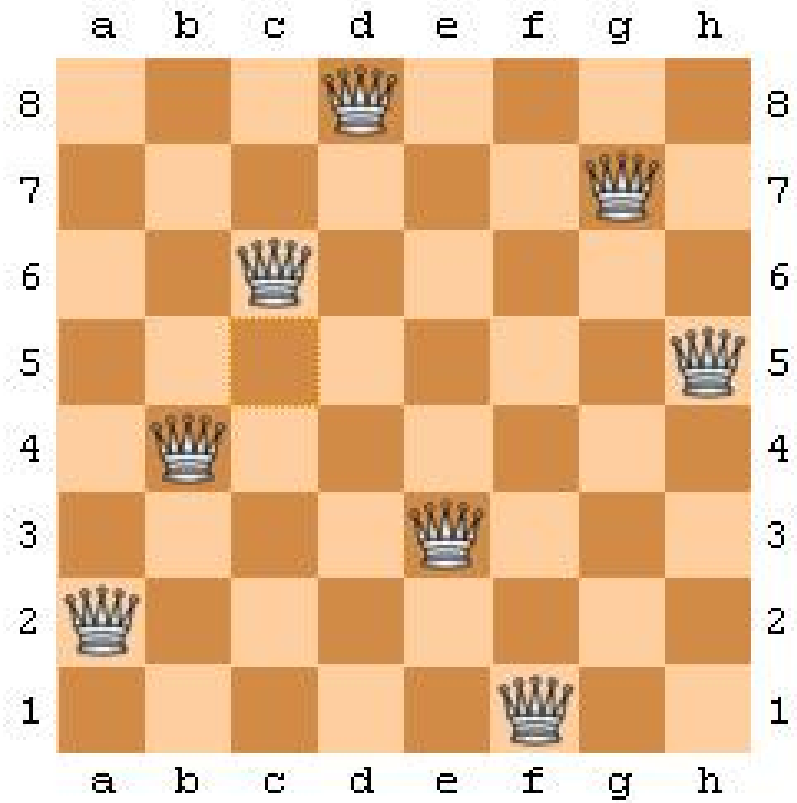
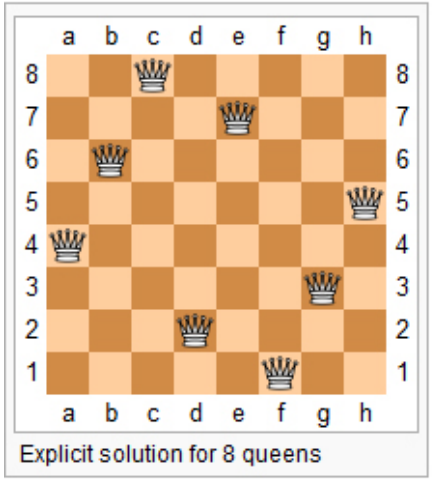
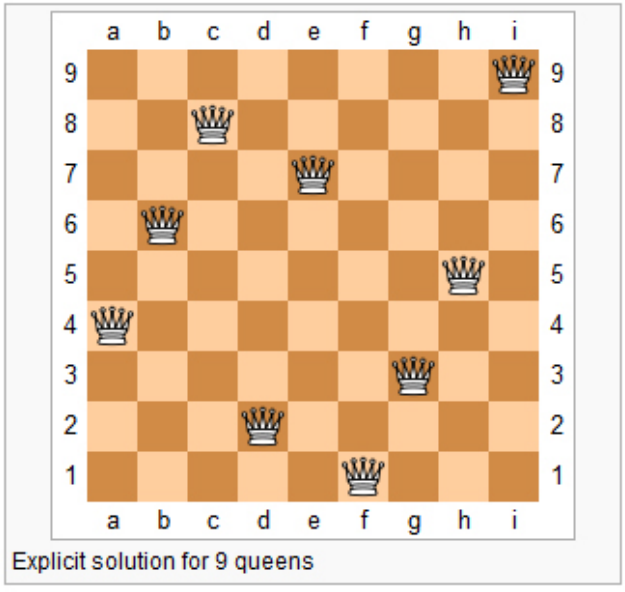


Imagem: http://software.intel.com/en-us/articles/multicore-enabling-the-n-queens-problem-using-cilk/

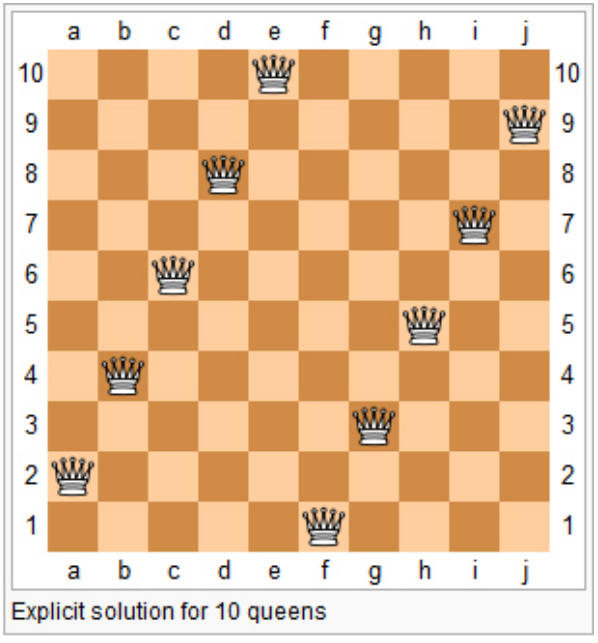
Soluções



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 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`.

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 $1, 2, \ldots, 8$

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

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.

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 um tabuleiro n × 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 i, int *s);
```

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;
```

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;
       nJogadas += 1;
       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 [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

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

Rainhas em uma mesma diagonal

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

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

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

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

Rainhas em uma mesma diagonal

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

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

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

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

ou seja

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

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 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.

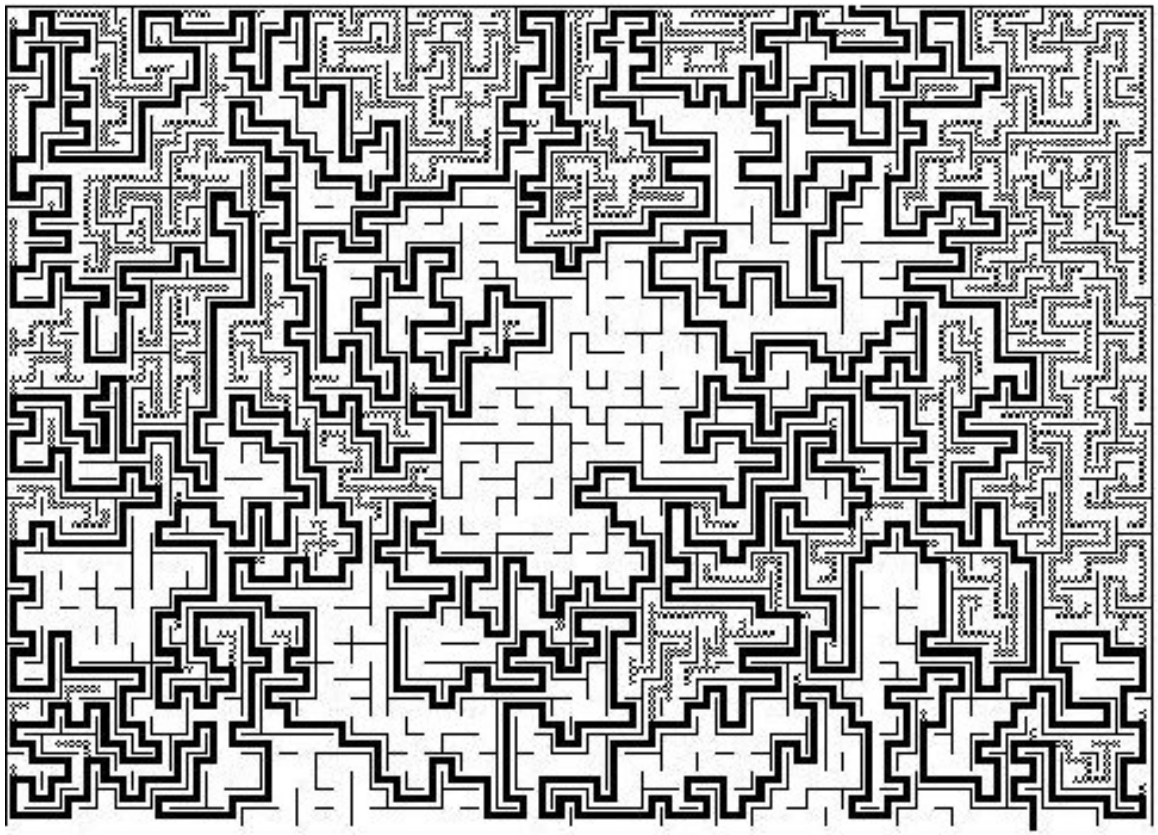
Backtracking

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*.

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:

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

Árvore de estados

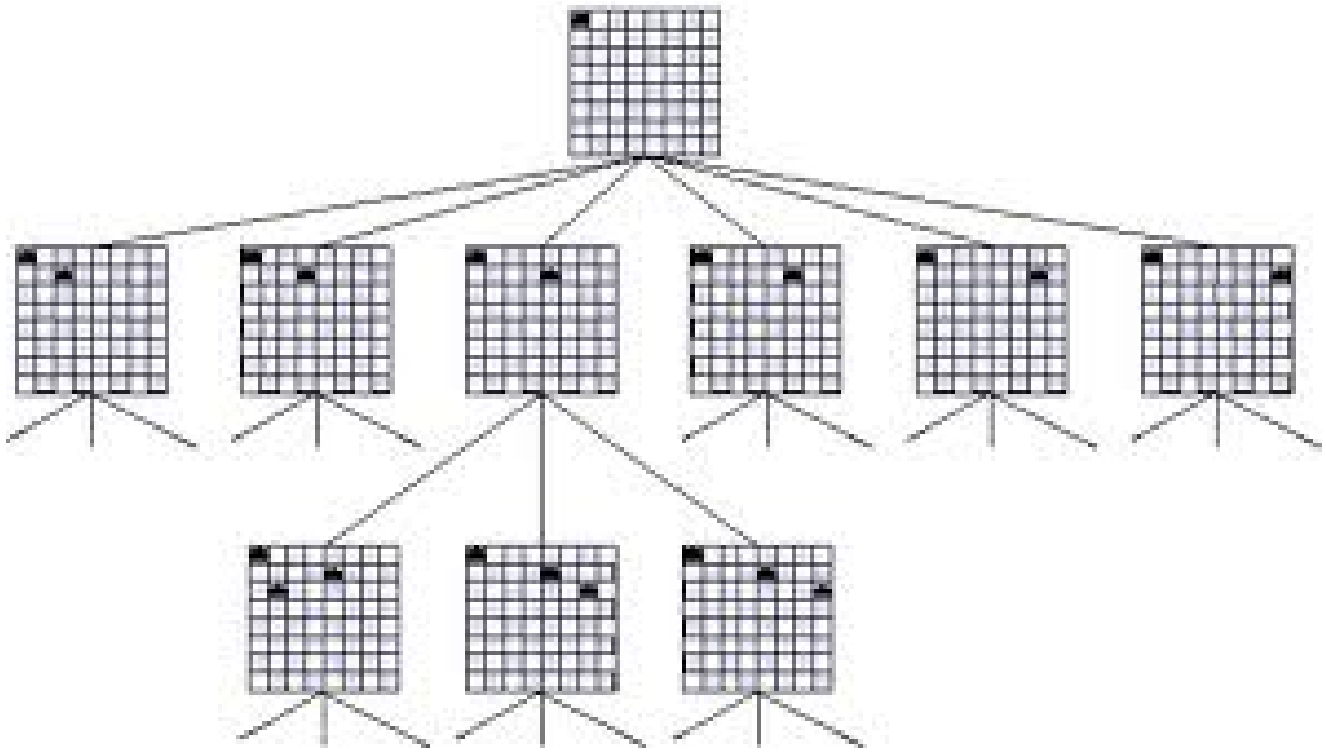


Imagem: http://cs.smith.edu/ thiebaut/transputer/chapter9/chap9-4.html

Árvore de estados

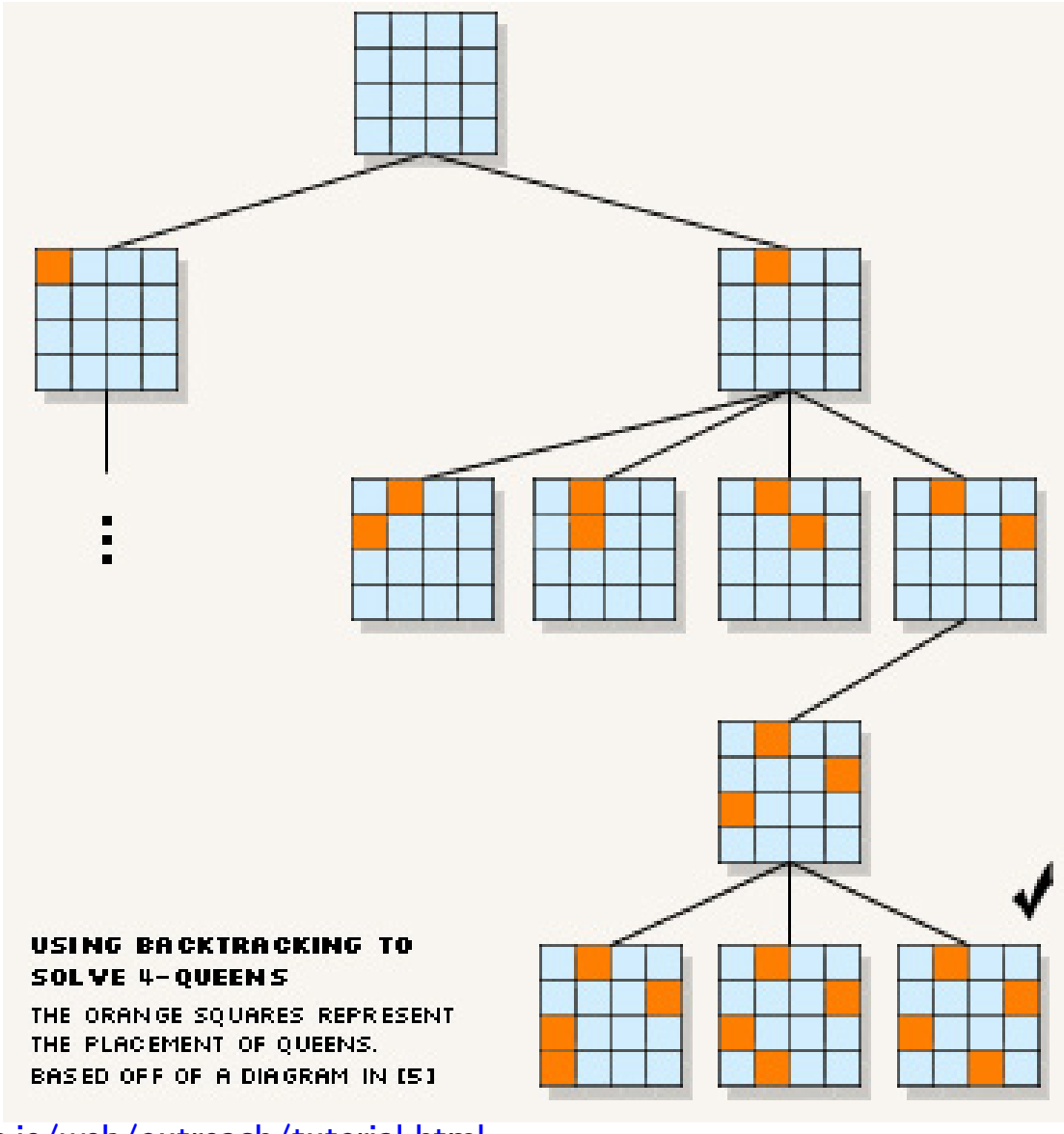


Imagem: http://4c.ucc.ie/web/outreach/tutorial.html

Função nRainhas (outra versão)

A função nRainhas a seguir imprime todas as configurações de n rainhas em 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) $s[1 \,..\, i-2]$ 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);
}
```