Problema das n rainhas

PF 12
http://www.ime.usp.br/~pf/algoritmos/aulas/enum.html
http://en.wikipedia.org/wiki/Eight_queens_puzzle

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) $s[1..i-1]$ 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 o tabuleiro com as rainhas em s[1..i] */
void
mostreTabuleiro(int n, int i, int *s);

/* Supondo 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 i; /* linha atual */
  int j; /* coluna candidata */
  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 (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(outra versão)

```
  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(outra versão)

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

Problema do passeio do cavalo

"*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: Suponha dado um tabuleiro de xadrez n-por-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.

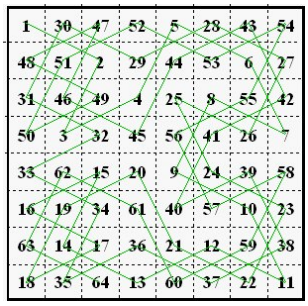

Imagem: http://www.magic-squares.net/knighttours.htm/

Soluções

	1	2	3	4	5
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

Soluções

	1	2	3	4	5	6
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

Soluções

	1	2	3	4	5	6	7
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

Movimentos do cavalo

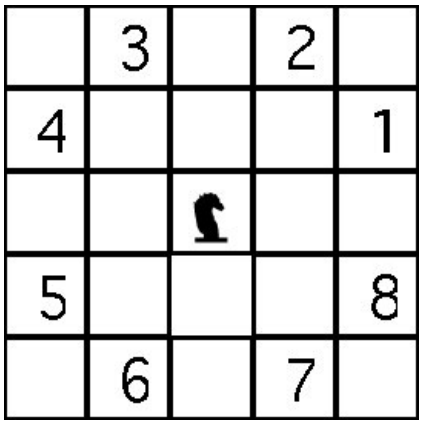

Imagem: http://www.mactech.com/articles/mactech/Vol.14/14.11/TheKnightsTour/index.html

Problema do passeio do cavalo

Cada movimento possível é de um dos tipos $1,\ldots,8$ 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

7.9×10^{50} seg $\approx 1.310^{49}$ min $\approx 2.1 \times 10^{47}$ 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 $n \times n$.

A mantém no início da cada iteração a seguinte relação invariante

(i0) $s[1..k-1]$ 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 as rainhas em s[1..i] */
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 mov */
  int *s = malloc((n*n+1)*sizeof(int));
  /* s[t] = movimento no passo t*/
  int k; /* passo atual */
```

Algoritmo passeioCavalo

```
  Movimento movimento[NMOV] = {
         {-1,+2},
         {-2,+1},
         {-2,-1},
         {-1,-2},
         {+1,-2},
         {+2,-1},
         {+2,+1},
         {+1,+2}
  };
  int mov;
```

Algoritmo passeioCavalo

```
/* linha 0 e coluna 0 do tabuleiro nao
serao usadas */
tab = malloc((n+1)*sizeof(int*));
for (i = 1; i <= n; i++)
   tab[i] = calloc(n+1,sizeof(int));
/* linha, coluna, movimento e tabuleiroa
inicial */
k = i = j = 1;
iProx = jProx = 0;
mov = 0;
tab[i][j] = 1;
```

Algoritmo passeioCavalo

```
while (0 < k && k < n*n) {
   int achouMov = FALSE;
   nMovimentos++;
   while(mov < NMOV && achouMov == FALSE){
      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 (mov < NMOV) { /* AVANCA */
      i = iProx;
      j = jProx;
      s[k] = mov;
      tab[i][j] = ++k;
      mov = 0;
   } 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");
   /* libera memoria alocada */
   free(s);
   for (i = 1; i <= n; i++)
      free(tab[i]);
   free(tab);
   printf("Num.  mov.=%d\n", nMovimentos);
}
```

Mais *backtracking*

O esquema a seguir tenta descrever o método *backtracking*.

Suponha que a solução de um problema pode ser vista como uma sequência de decisões

$$x[1], x[2], \ldots, x[n]$$

Por exemplo, cada $x[k]$ pode ser a posição de uma rainha ou para qual posição mover o cavalo.

A relação invariante chave do método é algo como

> *no início de cada iteração $x[1..k\text{-}1]$ é uma "solução parcial" (que pode ou não ser parte de uma solução)*

Mais *backtracking*

$k \leftarrow 1$
enquanto $k > 1$ **faça**
  procure um valor para $x[k]$ que ainda não foi testado e tal que $x[1..k]$ é solução parcial
  se encontrou candidato para $x[k]$
    então $k \leftarrow k + 1$ (**avança**)
      se $k = n$
        então encontramos uma solução
          devolva $x[1..n]$
          $k \leftarrow k - 1$ (**continua**)
    senão $k \leftarrow k - 1$ (**volta**)

Enumeração de subsequências

PF 12
http://www.ime.usp.br/~pf/algoritmos/aulas/enum.html

Enumeração de subsequências

Problema: Enumerar todas as subsequências de 1,2,...,n, ou seja, fazer uma lista em que cada subsequência aparece uma e uma só vez.
Exemplo: para n= 3 as subsequências são

```
1
1   2
1   2   3
1   3
2
2   3
3
```

Enumeração de subsequências

Exemplo: para n= 4 as subsequências são

```
1
1   2
1   2   3
1   2   3   4
1   2   4
1   3
1   3   4
1   4
2
2   3
2   3   4
2   4
3
3   4
4
```

subseqLex

A função subseqLex recebe n e imprime todas as subsequências não vazias de 1 . . n.

```
void subseqLex (int n) {
  int *s, k;
  s = mallocSafe((n+1) * sizeof(int));
  s[0] = 0;
  k = 0;
```

subseqLex

```
  while(1) {
      if (s[k] < n) {
          s[k+1] = s[k] + 1;
          k += 1;
      } else {
          s[k-1] += 1;
          k -= 1;
      }
      if (k == 0) break;
      imprima(s, k);
  }
  free(s);
}
```