Busca Completa

prof. Fábio Luiz Usberti

MC521 - Desafios de Programação I

Instituto de Computação - UNICAMP - 2018

Sumário

Busca Completa

- A busca completa é um método de solução de problemas que percorre todo o espaço de busca (de forma implícita ou explícita) para obter uma solução.
- Durante a busca é permitido (e recomendado) realizar podas em regiões do espaço de busca que não têm possibilidade de conter a solução desejada.
- Em programação competitiva, a busca completa deve ser a estratégia a ser adotada quando não há de modo claro um algoritmo alternativo, ou quando outros algoritmos existem, mas são de difícil implementação e desnecessários considerando o tamanho da entrada. Exemplo: o problema da consulta de mínimo em um intervalo ("Range Minimum Query" - RMQ) para entrada $N \leqslant 100$ pode ser resolvida utilizando busca completa, em tempo linear, para cada consulta.

Busca Completa

- Um algoritmo de busca completa nunca deve receber o veredito WA, dado que explora todo o espaço de busca. No entanto, a solução pode receber o veredito TLE.
- Ao fazer a análise de um algoritmo de busca completa, se essa análise indicar que o algoritmo deve passar na restrição de tempo, então não há porque não implementá-lo.
- Mesmo nos casos em que a busca completa não passa na restrição de tempo de execução, esse algoritmo pode ser utilizado como um verificador para instâncias pequenas, o que pode ser útil no desenvolvimento de um algoritmo mais eficiente.

Otimização de Código

A seguir serão fornecidas algumas dicas para otimização de códigos, especialmente úteis para algoritmos de busca completa.

- Acessar o conteúdo de um vetor bi-dimensional linha a linha ao invés de coluna a coluna, uma vez que os dados são armazenados sequencialmente em memória linha a linha.
- A manipulação de máscaras de bits ou de vetores do tipo `bitset` são mais eficientes do que `vector<bool>`.
- Se você tem a opção de escrever um código de modo iterativo ou recursivo, dê preferência à versão iterativa.
- O uso de vetor de caracteres é mais eficiente do que o tipo `string` em C++.

Busca Completa

- A seguir serão abordados problemas passíveis de serem tratados com busca completa.
- As soluções de cada problema não devem ser memorizadas, mas compreendidas.
- A proficiência na solução de novos problemas é muito mais relevante do que a memorização da solução de problemas em Ciência da Computação.

Busca Completa Iterativa

Dois Laços Aninhados

- Problema (UVa 725): Imprimir todos os pares de números de 5 dígitos que utilizam todos os dígitos de 0 a 9. O primeiro número dividido pelo segundo deve resultar em um valor inteiro N ($2 \leqslant N \leqslant 79$). Em outras palavras, queremos:

$$\frac{abcde}{fghij} = N$$

tal que cada letra representa um dígito distinto.

Busca Completa Iterativa

Dois Laços Aninhados

- Limitante superior: é possível notar os seguintes limitantes $01234 \leqslant abcde \leqslant 98765$ e $01234 \leqslant fghij \leqslant 98765$, o que corresponde a aproximadamente $100K$ possibilidades.
- Limitante superior 2: a partir da fração do enunciado é possível extrair um limitante superior mais apertado: $fghij \leqslant 98765/N$. Isso corresponde a menos do que $50K$ possibilidades, no pior caso onde $N = 2$.
- Estratégia: para cada possível valor de $fghij$, obtemos $abcde = fghij \times N$ e verificamos se todos os 10 dígitos são distintos. Logo o algoritmo corresponderá a um laço externo de $50K$ iterações e um laço interno de 10 iterações, o que corresponde a $500K$ operações, um valor relativamente baixo.

Busca Completa Iterativa

Dois Laços Aninhados

```
for (int fghij = 1234; fghij <= 98765 / N; fghij++) {
   int abcde = fghij * N; // this way, abcde and fghij are at most 5 digits
   int tmp, used = (fghij < 10000); // if digit f=0, then we have to flag it
   tmp = abcde; while (tmp) { used |= 1 << (tmp % 10); tmp /= 10; }
   tmp = fghij; while (tmp) { used |= 1 << (tmp % 10); tmp /= 10; }
   if (used == (1<<10) - 1)   // if all digits are used, print it
      printf("%0.5d / %0.5d = %d\n", abcde, fghij, N);
}
```

Busca Completa Iterativa

Múltiplos Laços Aninhados

- Problema (UVa 441): Dados k inteiros ($6 < k < 13$), enumere todos os subconjuntos de cardinalidade 6 ordenados de modo crescente. Os números inteiros já se encontram ordenados.
- Mesmo para o maior caso ($k = 12$), tem-se que o número de subconjuntos a serem impressos corresponde a $C_6^{12} = 924$, o que é um valor muito pequeno.
- Estratégia: busca completa utilizando 6 laços aninhados.

Busca Completa Iterativa

Múltiplos Laços Aninhados

```
for (int i = 0; i < k; i++)   // input: k sorted integers
   scanf("%d", &S[i]);
for (int a = 0; a < k - 5; a++)   // six nested loops!
   for (int b = a + 1; b < k - 4; b++)
      for (int c = b + 1; c < k - 3; c++)
         for (int d = c + 1; d < k - 2; d++)
            for (int e = d + 1; e < k - 1; e++)
               for (int f = e + 1; f < k; f++)
                  printf("%d %d %d %d %d %d\n",S[a],S[b],S[c],S[d],S[e],S[f]);
```

Laços com Podas

- Problema (UVa 11565): Dados três inteiros A, B e C ($1 \leqslant A, B, C \leqslant 10000$), encontre outros três números inteiros x, y e z, tais que:

$$x + y + z = A$$
$$xyz = B$$
$$x^2 + y^2 + z^2 = C$$
$$x < y < z$$

Busca Completa Iterativa

Laços com Podas

- A partir das equações do problema, é possível determinar intervalos de valores para x, y e z:
- Pela equação $x^2 + y^2 + z^2 = 10000$, temos que x, y e z estão no intervalo $[-100, 100]$.
- Pela equação $xyz = 10000$, temos que x está no intervalo $[-\lfloor\sqrt[3]{10000}\rfloor, \lfloor\sqrt[3]{10000}\rfloor] = [-21, 21]$.
- A partir desses três laços aninhados, teremos $43 * 121 * 120 \approx 620K$ operações, o que é plenamente viável.

Busca Completa Iterativa

Laços com Podas

```
bool sol = false; int x, y, z;
for (x = -21; x <= 21 && !sol; x++) if (x * x <= C)
   for (y = x+1; y <= 100 && !sol; y++) if (x * x + y * y <= C)
      for (z = y+1; z <= 100 && !sol; z++)
         if (x + y + z == A && x * y * z == B && x * x + y * y + z * z == C) {
            printf("%d %d %d\n", x, y, z);
            sol = true;
         }
```

Busca Completa Iterativa

Permutações

- Problema (UVa 11742): Há $1 \leqslant n \leqslant 8$ pessoas que sentam-se em uma mesma fileira de n assentos consecutivos. Existem $0 \leqslant m \leqslant 20$ restrições de ocupação dos assentos, que são da seguinte forma: os indivíduos a e b devem sentar-se a pelo menos k assentos de distância. Quantas permutações de assentos são viáveis para as restrições impostas?
- Estratégia: Para resolver esse problema, é possível explorar todas as possíveis permutações e enumerar quantas são viáveis.
- Considerando que a verificação de viabilidade de uma permutação requer uma ordem de $m \times n$ operações, temos que a complexidade do algoritmo será $O(m \times n \times n!)$.
- Considerando a maior instância, teremos $20 \times 8 \times 8! \approx 6M$ operações, o que deve respeitar a restrição de tempo.

Busca Completa Iterativa

Permutações

```
#include <algorithm>   // next_permutation is inside this C++ STL
// the main routine
int i, n = 8, p[8] = {0, 1, 2, 3, 4, 5, 6, 7};   // the first permutation
do {     // try all possible O(n!) permutations, the largest input 8! = 40320
    ...  // check the given social constraint in O(m*n)
}        // the overall time complexity is O(m*n*n!)
while (next_permutation(p, p + n));   // this is inside C++ STL <algorithm>
```

Busca Completa Iterativa

Subconjuntos

- Problema (UVa 12455): Dado um conjunto de $1 \leqslant n \leqslant 20$ números inteiros, existe algum subconjunto tal que a soma de seus números resulta no valor X?
- Estratégia: avaliar todos os 2^n subconjuntos, somando os valores de cada conjunto em $O(n)$.
- Esse algoritmo apresenta uma complexidade $O(n \cdot 2^n)$.
- Para a maior instância ($n = 20$), tem-se na ordem de $20 \times 2^{20} \approx 21M$, o que está próximo do limiar do número de execuções efetuadas por segundo em uma máquina moderna.
- Para que o algoritmo de busca completa não ultrapasse o tempo limite, as operações devem ser implementadas de modo eficiente, o que é possível utilizando uma máscara de bits para representar os subconjuntos.

Busca Completa Iterativa

Subconjuntos

```
// the main routine, variable 'i' (the bitmask) has been declared earlier
for (i = 0; i < (1 << n); i++) { // for each subset, O(2^n)
   sum = 0;
   for (int j = 0; j < n; j++)   // check membership, O(n)
      if (i & (1 << j))          // test if bit 'j' is turned on in subset 'i'?
         sum += l[j];            // if yes, process 'j'
   if (sum == X) break;          // the answer is found: bitmask 'i'
}
```

Busca Completa Recursiva

Problema das 8-rainhas

- Problema (UVa 00750): Em um tabuleiro 8×8 de xadrez, é possível posicionar oito rainhas de tal modo que nenhum par de rainhas se ataque. Determine todos os possíveis arranjos considerando que a posição de uma das rainhas é fornecida.
- Estratégia 1: A estratégia mais ingênua consiste em enumerar todas as possiveis $C_8^{64} \approx 4B$ combinações de rainhas no tabuleiro e verificar, uma combinação por vez, se as rainhas encontram-se em posição de ataque.
- Estratégia 2: Uma estratégia melhor consiste em utilizar o fato de que cada linha e coluna é ocupada por no máximo uma rainha. Isso reduz o problema original para o problema de encontrar uma permutação válida de linhas (vetor `row`) do tabuleiro, tal que `row[i]` descreve em que linha se encontra a rainha posicionada na i-ésima coluna.

Problema das 8-rainhas

- A complexidade do algoritmo correspondente à segunda estratégia é fatorial $O(n!)$, no entanto o valor de n é pequeno e o número de operações será da ordem de $8! \approx 40K$.
- Há também o fato de que duas rainhas não compartilham diagonais. Isso pode ser verificado da seguinte forma: duas rainhas A e B, com coordenadas (i, j) e (k, l), respectivamente, encontram-se na mesma diagonal se e somente se $abs(i - k) == abs(j - l)$.

Busca Completa Recursiva

Problema das 8-rainhas (Continua...)

```
/* 8 Queens Chess Problem */
#include <cstdlib>                          // we use the int version of 'abs'
#include <cstdio>
#include <cstring>
using namespace std;

int row[8], TC, a, b, lineCounter;              // ok to use global variables

bool place(int r, int c) {
  for (int prev = 0; prev < c; prev++)     // check previously placed queens
    if (row[prev] == r || (abs(row[prev] - r) == abs(prev - c)))
      return false;          // share same row or same diagonal -> infeasible
  return true; }

void backtrack(int c) {
  if (c == 8 && row[b] == a) {           // candidate sol, (a, b) has 1 queen
    printf("%2d      %d", ++lineCounter, row[0] + 1);
    for (int j = 1; j < 8; j++) printf(" %d", row[j] + 1);
    printf("\n"); }
  for (int r = 0; r < 8; r++)                          // try all possible row
    if (place(r, c)) {            // if can place a queen at this col and row
      row[c] = r; backtrack(c + 1);       // put this queen here and recurse
}   }
```

Busca Completa Recursiva

Problema das 8-rainhas

```
int main() {
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &a, &b); a--; b--;          // switch to 0-based indexing
    memset(row, 0, sizeof row); lineCounter = 0;
    printf("SOLN       COLUMN\n");
    printf(" #      1 2 3 4 5 6 7 8\n\n");
    backtrack(0);                // generate all possible 8! candidate solutions
    if (TC) printf("\n");
} } // return 0;
```

Busca Completa Recursiva

Problema das n-rainhas

- Problema (UVa 11195): Considere um tabuleiro $n \times n$ de xadrez ($3 < n < 15$), tal que algumas das casas (células) estão impedidas (não pode conter uma rainha). De quantas maneiras distintas é possível posicionar n rainhas de modo que nenhum par de rainhas se ataque.
- O algoritmo para o problema anterior (UVa 00750) não é rápido o suficiente para $n = 14$. A maior ineficiência do algoritmo anterior consiste na validação da posição de uma nova rainha, pois testa-se com todas as $c - 1$ posições de rainhas já alocadas.

Busca Completa Recursiva

Problema das *n*-rainhas

- Estratégia: o diferencial do novo algoritmo será armazenar as linhas, colunas e diagonais ocupadas utilizando três vetores de bits: `rw` (linhas), `ld` (colunas a esquerda), `rw` (colunas a direita).

```
bitset<30> rw, ld, rd; // for the largest n = 14, we have 27 diagonals
```

- Quando uma rainha é alocada na linha r, configura-se `rw[r] = true` para impedir que essa linha seja utilizada novamente.
- Com relação às diagonais, quando uma rainha é alocada na posição (r, c), também são configurados `ld[r - c + n - 1] = true` e `rd[r + c] = true`.
- Finalmente, as células impedidas são armazenadas em uma matriz `board[][]`.

Busca Completa Recursiva

Problema das *n*-rainhas

```
void backtrack(int c) {
   if (c == n) { ans++; return; }     // a solution
   for (int r = 0; r < n; r++)        // try all possible row
      if (board[r][c] != '*' && !rw[r] && !ld[r - c + n - 1] && !rd[r + c]) {
         rw[r] = ld[r - c + n - 1] = rd[r + c] = true;      // flag
         backtrack(c + 1);
         rw[r] = ld[r - c + n - 1] = rd[r + c] = false;     // restore
}  }
```

Referências

1. S. Halim e F. Halim. Competitive Programming 2, Second Edition Lulu (www.lulu.com), 2011. (IMECC – 005.1 H139c)
2. S. S. Skiena, M. A. Revilla. Programming Challenges: The Programming Contest Training Manual, Springer, 2003.
3. T.H. Cormen, C.E. Leiserson, R.L.Rivest e C. Stein. Introduction to Algorithms. 2nd Edition, McGraw-Hill, 2001. (no. chamada IMECC – 005.133 Ar64j 3.ed.)
4. U. Manber. Introduction to Algorithms: A Creative Approach. Addison-Wesley. 1989. (no. chamada IMECC – 005.133 Ec53t 2.ed.)