Programação Dinâmica

prof. Fábio Luiz Usberti

MC521 - Desafios de Programação I

Instituto de Computação - UNICAMP - 2018

Sumário

Introdução

- Serão investigados problemas que apresentam soluções por programação dinâmica.
- Os estados e transições em programação dinâmica são bem conhecidos para esses problemas.
- Os problemas investigados são:

1. Máxima Soma de Intervalo (*Max Range Sum*)
2. Maior Subsequência Crescente (*Longest Increasing Subsequence*)
3. Problema do Troco em Moedas (*Coin Change Problem*)
4. Problema do Caixeiro Viajante (*Traveling Salesman Problem*)

Máxima Soma de Intervalo

UVa 507 – Jill Rides Again

- Enunciado: Dado um vetor $A[1..n]$ contendo $0 \leqslant n \leqslant 20K$ valores inteiros, determine um intervalo definido por dois índices i e j, maximizando a soma $A[i] + \ldots + A[j]$, ou seja:

$$RSQ^*[A] = \max_{i,j} \quad RSQ[i,j]$$

tal que $RSQ^*[A]$ é a solução ótima para o vetor A e

$$RSQ[i,j] = \sum_{k=i}^{j} A[k]$$

- Observação: Intervalos vazios são permitidos, portanto a solução ótima para um vetor contendo somente números negativos é $RSQ^*[A] = 0$.

Problema 1

Máxima Soma de Intervalo

- Solução por Busca Completa 1: Um algoritmo de busca completa avalia $RSQ[i,j]$ ($O(n)$) para todos os pares (i,j) ($O(n^2)$), retornando o valor máximo encontrado em tempo $O(n^3)$.

Problema 1

Máxima Soma de Intervalo

- Solução por Busca Completa 2: É possível realizar um pré-processamento, criando um vetor de acumulação $B[0..n]$, da seguinte forma:

$$B[i] = \begin{cases} 0 & i = 0 \\ A[i] + B[i-1] & i > 0 \end{cases}$$

- Nesse caso, a avaliação $RSQ[i,j]$ pode ser feita em tempo $O(1)$, pois $RSQ[i,j] = B[j] - B[i-1]$ (para $i \leqslant j$). Logo a complexidade da busca completa passa a ser limitada por $O(n^2)$.
- Para $n = 20K$, mesmo a complexidade $O(n^2)$ resulta em veredito TLE.

Problema 1

Máxima Soma de Intervalo

- Um algoritmo de programação dinâmica proposto por Jay Kadane, possui complexidade linear $O(n)$, o que é assintoticamente ótimo.
- A ideia principal do algoritmo consiste em acumular a soma dos inteiros enquanto esse valor for maior que 0. Se o acúmulo resultar em um valor negativo, a soma é reinicializada em 0.
- A subestrutura ótima utilizada por Kadane considera um vetor $dp[0..n]$, tal que $dp[i]$ contém a soma do melhor intervalo que termina em $A[i]$. Cabe lembrar que como intervalos vazios são permitidos, então $dp[i] = 0$ no pior caso.

$$dp[i] = \begin{cases} 0 & i = 0 \\ \max[0, A[i] + dp[i-1]] & i > 0 \end{cases}$$

Problema 1

Máxima Soma de Intervalo

- O valor da solução ótima pode ser obtida a partir do maior valor armazenado no vetor $dp[i]$.

$$RSQ^*[A] = \max_i \; dp[i]$$

Exemplo

- $A = [4, -5, 4, -3, 4, 4, -4, 4, -5]$
- $dp = [0, 4, 0, 4, 1, 5, 9, 5, 9, 4]$

Implementação Bottom-Up

```
using namespace std;

int A[MAX_N], DP[MAX_N+1];

/* maximum range sum algorithm */
int dp_mrs(int n) {

  DP[0] = 0;
  int maxMRS = DP[0];

  for (int i = 0; i < n; ++i) {
    DP[i+1] = max(0,DP[i]+A[i]);
    maxMRS = max(maxMRS,DP[i+1]);
  }

  return maxMRS;
}
```

Problema 2

Maior Subsequência Crescente (Longest Increasing Subsequence)

- Dado um vetor $A[1..n]$, encontre a maior subsequência crescente de elementos (LIS^*).
- Cabe lembrar que em uma subsequência os elementos não precisam ser consecutivos.

Exemplo:

- $A = [-7, 10, 9, 2, 3, 8, 8, 1]$
- $LIS^*(A) = 4$ correspondente à sequência de quatro elementos $[-7, 2, 3, 8]$

Problema 2

Maior Subsequência Crescente (Longest Increasing Subsequence)

- Um algoritmo de busca completa consiste em explorar todas as possíveis subsequências, o que é ineficiente, dado que a quantidade de subsequências é da ordem de $O(2^n)$.
- Em uma abordagem por programação dinâmica, consideramos uma função $LIS(i)$ que retorna o tamanho da maior subsequência crescente que termina com o elemento $A[i]$.

Problema 2

Maior Subsequência Crescente (Longest Increasing Subsequence)

- Caso base: $LIS(1) = 1$.
- Caso geral: $LIS(i)$, para $i > 0$, consiste em encontrar um índice $j < i$ tal que $A[j] < A[i]$ e $LIS[j]$ seja máximo. Se tal índice j existir $LIS(i) = LIS(j) + 1$, caso contrário, $LIS(i) = 1$.

$$LIS[i] = \begin{cases} 1 & i = 1 \\ 1 + \max\limits_{j} \ (LIS[j]) & i > 1, j < i, A[j] < A[i] \end{cases}$$

Maior Subsequência Crescente (Longest Increasing Subsequence)

- O valor da solução ótima será $LIS^*[A] = \max_k LIS[k]$.
- A abordagem por programação dinâmica aproveita-se do fato desse problema possuir sobreposição de subproblemas, dado que são calculados $LIS(j)$ para $\forall j \in [0..i-1]$.
- O número de estados é igual a n e cada estado é computado em $O(n)$, logo a complexidade do algoritmo de programação dinâmica é limitada por $O(n^2)$.

Exemplo:

Index	0	1	2	3	4	5	6	7
A	-7	10	9	2	3	8	8	1
LIS(i)	1	2	2	2	3	4	4	2

Maior Subsequência Crescente

Implementação Bottom-Up

```
#define MAX_N 10000

int A[MAX_N], DP[MAX_N];

/* longest increasing subsequence algorithm */
int dp_lis(int n) {

  DP[0] = 1;
  int maxLIS = DP[0];

  for (int i = 1; i < n; ++i) {
     DP[i] = 1;
     for (int j = 0; j < i; ++j)   {
        if (A[j] < A[i]) {
           DP[i] = max(DP[i],DP[j]+1);
        }
     }

    maxLIS = max(maxLIS,DP[i]);
  }

  return maxLIS;
}
```

Problema do Troco em Moedas (*Coin Change Problem*)

- Problema: Dado um valor V e uma lista de n denominações de moedas ($coin[i], \quad i \in [1, \ldots, n]$), qual o número mínimo de moedas que devemos utilizar para somar o valor V? Pode-se assumir que há quantidades ilimitadas de cada denominação de moeda.
- A estratégia gulosa é válida somente para algumas denominações de moedas.
- Para o caso geral é possível aplicar um algoritmo de programação dinâmica, com a seguinte subestrutura ótima:

$$dp[V] = \begin{cases} \infty & V < 0 \\ 0 & V = 0 \\ 1 + \min_i dp[V - coin[i]] & V > 0 \end{cases}$$

- Onde $dp[V]$ consiste na quantidade mínima de moedas necessárias para atingir um valor V.

Problema 3

Problema do Troco em Moedas (*Coin Change Problem*)

Exemplo:

<0	0	1	2	3	4	5	6	7	8	9	10
∞	0	1	2	3	4	1	2	3	4	5	2

V = 10, N = 2, coinValue= {1, 5}

- O problema possui $V+1$ estados possíveis e o cálculo de cada estado requer um processamento da ordem de $O(n)$, portanto a complexidade do algoritmo é limitada por $O(nV)$.

$$dp[V] = \begin{cases} \infty & V < 0 \\ 0 & V = 0 \\ 1 + \min_i dp[V - coin[i]] & V > 0 \end{cases}$$

Problema do Troco em Moedas

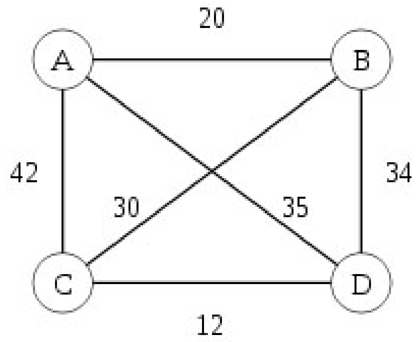
Implementação Bottom-Up

```
#define MAX_N 10000
#define MAX_VAL 1000000

int COIN[MAX_N], DP[MAX_VAL];

/* coin change dp algorithm
   val - value of the coin change
   n - number of coin denominations */
int dp_coin(int val, int n) {

  DP[0] = 0;

  for (int v = 1; v <= val; ++v) {
     DP[v] = MAX_VAL;
     for (int i = 0; i < n; ++i)   {
        if (v >= COIN[i]) {
           DP[v] = min(DP[v],DP[v-COIN[i]]+1);
        }
     }

  }

  return DP[val];
}
```

Problema 4

Problema do Caixeiro Viajante (*Traveling Salesman Problem (TSP)*)

- Problema: Considere n cidades para as quais uma matriz de distâncias é fornecida. Deseja-se encontrar uma rota de mínima distância que visita todas as cidades.

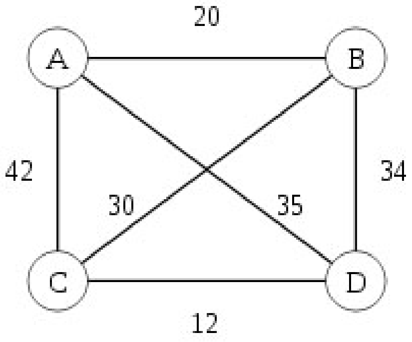


	A	B	C	D
A	0	20	42	35
B	20	0	30	34
C	42	30	0	12
D	35	34	12	0

Problema 4

Problema do Caixeiro Viajante (*Traveling Salesman Problem (TSP)*)

- Um algoritmo de busca completa deve explorar $(n-1)!/2$ rotas distintas.
- Para o exemplo abaixo há 3 soluções distintas, dentre as quais a solução ótima $A-B-C-D-A$, cujo valor é $20+30+12+35=97$.



	A	B	C	D
A	0	20	42	35
B	20	0	30	34
C	42	30	0	12
D	35	34	12	0

Problema 4

Problema do Caixeiro Viajante (*Traveling Salesman Problem (TSP)*)

- O TSP possui sobreposição de subproblemas. Para visualizar essa propriedade, imagine uma solução que passa pelas cidades $A - B - C - D - ... - X - ... - A$, onde X é uma subrota de $n - 3$ cidades.
- Agora considere a solução alternativa $A - C - B - D - ... - X - ... - A$.
- As duas soluções acima possuem sobreposição de subproblemas uma vez que ambas recaem no mesmo subproblema que consiste em obter a melhor permutação de X dado que a última cidade visitada foi D.
- A programação dinâmica aproveita-se dessas sobreposições, calculando uma única vez o custo de cada subrota.

Problema 4

Problema do Caixeiro Viajante (*Traveling Salesman Problem (TSP)*)

- Para representar um subproblema (estado) serão utilizadas duas informações: a última cidade visitada i e o subconjunto de cidades já visitadas S.
- Uma boa maneira de representar o conjunto S é através de uma máscara de bits. Para uma instância com n cidades serão necessários n bits. Uma cidade k foi visitada se e somente se o k-ésimo bit menos significativo for igual a 1.
- Exemplo: $S = 19_{10} = 10011_2$ implica que as cidades 0, 1 e 4 foram visitadas, i.e., pertencem ao conjunto S.

Problema do Caixeiro Viajante (*Traveling Salesman Problem (TSP)*)

- Seja S o conjunto de cidades já visitadas e $i \in S$ a última cidade visitada. Seja também $tsp[i, S]$ o custo da subrota ótima que percorre as cidades ainda não visitadas e retorna ao nó inicial 0.
- A subestrutura ótima é dada por:

$$tsp[i,S] = \begin{cases} dist(i,0) & S = 2^n - 1 \\ \min\limits_{j \in [0..n-1] \setminus S} [dist(i,j) + tsp(j, S \cup \{j\})] & \text{c.c.} \end{cases}$$

Problema 4

Problema do Caixeiro Viajante (*Traveling Salesman Problem (TSP)*)

$$tsp[i,S] = \begin{cases} dist(i,0) & S = 2^n - 1 \\ \min\limits_{j \in [0..n-1] \setminus S} [dist(i,j) + tsp(j, S \cup \{j\})] & \text{c.c.} \end{cases}$$

- A quantidade de possíveis subproblemas a serem avaliados é da ordem de $O(n2^n)$, pois $i \in [0..n-1]$ e $S \in [0..2^n - 1]$.
- Como o cálculo de um estado exige na ordem de $O(n)$ chamadas recursivas (transições de estado), a complexidade assintótica do algoritmo passa a ser $O(n^2 2^n)$.
- Com o algoritmo de programação dinâmica, o número de cidades que podem ser resolvidas em um ambiente competitivo é de aproximadamente 16 cidades.

Problema do Caixeiro Viajante (*Traveling Salesman Problem (TSP)*)

Implementação Top-Down

```
/* distances matrix and memoization table */
int dist[20][20], memo[20][1 << 20];

/* dp top-down algorithm for TSP */
int tsp(int pos, int bitmask) { // bitmask stores the visited coordinates
  if (bitmask == (1 << n)) - 1)
    return dist[pos][0]; // return trip to close the loop
  if (memo[pos][bitmask] != -1)
    return memo[pos][bitmask];

  int ans = 2000000000;
  for (int nxt = 0; nxt < n; nxt++) // O(n) here
    if (nxt != pos && !(bitmask & (1 << nxt))) // if coordinate nxt is not visited yet
      ans = min(ans, dist[pos][nxt] + tsp(nxt, bitmask | (1 << nxt)));
  return memo[pos][bitmask] = ans;
}
```

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.)