算法设计与分析复习参考

沈容舟 anticlockwise5@gmail.com

2006年6月6日

Contents

1 O, Ω, Θ 的具体定义 **3**

2 最坏情况下复杂度和平均情况下复杂度 **3**

3 递归 **3**
- 3.1 定义 3
- 3.2 递归算法二要素 3

4 背包问题与0-1背包问题的区别 **3**

5 分治法与动态规划之间的异同 **3**

6 回溯分治法与分支限界法的异同 **4**

7 动态规划问题的特征 **4**

8 回溯法中的剪枝函数 **4**

9 非确定性图灵机与确定性图灵机的主要区别 **4**

10 P类与NP类语言 **4**
- 10.1 P类和NP类语言的定义 4

11 旅行售货员问题 **5**
- 11.1 算法描述 5
  - 11.1.1 队列式分支限界法 5
  - 11.1.2 优先队列式分支限界法 5

12 0-1 背包问题 **5**
- 12.1 最优子结构性质 6
- 12.2 递归关系 6
- 12.3 划分阶段 6
- 12.4 确定装入 7
- 12.5 例题 7
- 12.6 算法实现 7
  - 12.6.1 动态规划求最优值 7
  - 12.6.2 构造最优解 7
- 12.7 算法复杂度分析 8

13 n后问题 8
13.1 算法描述 8
13.1.1 解向量 8
13.1.2 显约束 8
13.1.3 隐约束 8

14 图的m着色问题 9
14.1 算法描述 9
14.1.1 解空间 9
14.1.2 可行性约束函数 9
14.2 算法复杂度 10

15 最后一道设计题 10
15.1 思路 10
15.2 算法复杂度分析 10

1 O, Ω, Θ 的具体定义

以下设f(N)和g(N)是定义在正数集上的正函数。

如果存在正的常数C和自然数N_0，使得当N≥N_0时有f(N)≤Cg(N)，则称 函数f(N)当N充分大时上有界，且g(N)是它的一个上界，记为f(N)=O(g(N))。这时 还说f(N)的阶不高于g(N)的阶。

如果存在正的常数C和自然数N_0，使得当N≥N_0时，有f(N)≥Cg(N)， 则称函数f(N)当N充分大时下有界；且g(N)是它的一个下界，记为f(N)=Ω(g(N))。

2 最坏情况下复杂度和平均情况下复杂度

1. 最坏情况下的时间复杂度：$T\max(n) = \max\{T(N,I)|size(I)=n\}$
2. 最好情况下的时间复杂度：$T\min(n) = \min\{T(N,I)|size(I)=n\}$
3. 平均情况下的时间复杂度：$Tavg(n) = \sum\limits_{size(I)=n} p(I)T(I)$

3 递归

3.1 定义

直接或间接的调用自身的算法称为**递归算法**。用函数自身给出定义的 函数称为**递归函数**。

3.2 递归算法二要素

1. 递归边界条件。
2. 递归定义：使问题向边界条件转化的规则。

4 背包问题与0-1背包问题的区别

- 共同点：给定n种物品和一个背包。物品i的重量是W_i，其价值为V_i，背包的容量为C。 应如何选择装入背包的物品，使得装入背包种物品的总价值最大。
- 不同点：
  1. 对于0-1背包问题，在选择装入背包的物品时，对每种物品i只有2种选择， 即装入背包或不装入背包。不能将物品i装入背包多次，也不能只装入部分的物品i。
  2. 对于背包问题，物品装入时，可以装入部分物品。

5 分治法与动态规划之间的异同

- 相同点：基本思想相同，即都是将待求解问题分解成若干个子问题，先求解自问题，然后从这些 子问题的解得到原问题的解。
- 不同点：适合于用动态规划求解的问题，经分解得到的子问题往往不是互相对立的。

6 回溯分治法与分支限界法的异同

- 相同点：都是在问题的解空间上搜索问题解得算法
- 不同点：

1. 求解目标区别：回溯法的求解目标是找出解空间树种满足约束条件的**所有解**， 而分支限界法的求解目标则是找出满足约束条件的**一个解**，或是在满足约束条件的 解中找出在某种意义下的**最优解**。
2. 搜索方式不同：回溯法以深度优先的方式搜索解空间树，而分支限界法则以广度 优先或以最小耗费优先的方式搜索解空间树。

7 动态规划问题的特征

- 最优子结构性质：问题的最优解包含了其子问题的最优解。
- 子问题重叠性质：在求解问题时，每次产生的子问题并不一定完全是新的或者独立的问题， 而有一些子问题是重复计算的。

8 回溯法中的剪枝函数

- 约束函数：用来在扩展结点处剪去不满足约束的子树；
- 限界函数：用来剪去得不到最优解的子树。

9 非确定性图灵机与确定性图灵机的主要区别

- 确定性图灵机：在图灵机计算模型中，移动函数δ是单值的，即对于$Q \times Tk$中的每一个值， 当它属于δ的定义域时，$Q \times (T \times \{L, R, S\})k$中只有唯一的值与之对应， 称这种图灵机为确定性图灵机，简记为DTM(Deterministic Turing Machine)。
- 非确定性图灵机：一个k带的非确定性图灵机M是一个7元组：(Q, T, I, δ, b, q_0, q_f)。 与确定性图灵机不同的是非确定性图灵机允许移动函数δ具有不确定性，即对于$Q \times Tk$中的每一个 值$(q; x_1, x_2, \ldots, x_k)$，当它属于$\delta$的定义域时，$Q \times (T \times \{L, R, S\})k$中有唯一的一个子集$\delta(q; x_1, x_2, \ldots, x_k)$与之对应。可以在$\delta(q; x_1, x_2, \ldots, x_k)$中随意选定一个值作为他的函数值。

10 P类与NP类语言

问题P是多项式（时间）界的，是指问题P有一个多项式（时间）界的算法。 问题P一般称为P类问题，或易解问题。

一般的说，将可由多项式时间算法求解的问题看作是**易处理的问题**， 而将需要超多项式时间才能求解的问题看作是**难处理的问题**。

10.1 P类和NP类语言的定义

$P = \{$L|L是一个能在多项式时间内被一台DTM所接受的语言$\}$
$NP = \{$L|L是一个能在多项式时间内被一台NDTM所接受的语言$\}$

11 旅行售货员问题

11.1 算法描述

算法中while循环的终止条件是排列树的一个叶结点成为当前扩展结点。 当s=n-1时，以找到的回路前缀是$x[0 : n-1]$，它已包含图G的所有n个顶点。 因此，当s=n-1时，相应的扩展结点表示一个叶结点。 此时该叶结点所相应的回路的费用等于cc和lcost的值。 剩余的活结点的lcost值不小于以找到的回路的费用。 它们都不可能导致费用更小的回路。 因此已找到的叶结点所相应的回路是一个最小费用旅行售货员回路，算法可以结束。

算法结束时返回找到的最小费用，相应的最优解有数组v给出。

11.1.1 队列式分支限界法

$[\underline{A}]B, C, D$
$[\underline{B}, C, D]E, F$
$[\underline{C}, D, E, F]G, H$
$[\underline{D}, E, F, G, H]I, J$
$[\underline{E}, F, G, H, I, J]K(59)[1, 2, 3, 4]$
$[\underline{F}, G, H, I, J]L(66)$
$[\underline{G}, H, I, J]M(25)[1, 3, 2, 4]$
$[\underline{H}, I, J]1-3-4(26)$
$[\underline{I}, J]O(25)$
$[\underline{J}]P(59)$

11.1.2 优先队列式分支限界法

$[\underline{A}]B, C, D \to B(30), C(6), (4)$
$[\underline{D}, C, B]I, J \to I(14), J(24)$
$[\underline{C}, I, J, B]G, H \to G(11), H(26)$
$[\underline{G}, I, J, B, H]M \to M(25)[1, 2, 3, 4]$
$[\underline{I}, J, B, H]O \to O(25)$
$[\underline{J}, B, H]P \to P(59)$
$[\underline{B}, H]B, H$限界掉

12 0-1 背包问题

0-1背包问题：给定n种物品和一个背包。物品i的重量是w_i， 其价值为v_i，背包的容量为C。问：应该如何选择装入背包的物品， 使得装入背包种物品的总价值最大？

在选择装入背包的物品时，对每种物品i只有两种选择，即装入背包或不装入背包。 不能将物品i装入背包多次，也不能只装入部分的物品i。

12.1 最优子结构性质

0-1背包问题具有最优子结构性质。设$(y_1, y_2, \ldots, y_n)$是所给0-1背包问题的一个最优解， 则$(y_2, \ldots, y_n)$是下面相应子问题的一个最优解：

$$\max \sum_{i=2}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=2}^{n} w_i x_i \le C \\ x_i \in \{0, 1\}, 2 \le i \le n \end{cases}$$

因若不然，设$(z_2, \ldots, z_n)$是上述子问题的一个最优解， 而$(y_2, \ldots, y_n)$不是它的最优解。 由此可知，$\sum_{i=2}^{n} v_i z_i > \sum_{i=2}^{n} v_i y_i$， 且$w_1 y_1 + \sum_{i=2}^{n} w_i z_i \le C$。因此

$$v_1 y_1 + \sum_{i=2}^{n} v_i z_i > \sum_{i=2}^{n} v_i y_i$$

$$w_1 y_1 + \sum_{i=2}^{n} w_i z_i \le C$$

这说明$(z_1, z_2, \ldots, z_n)$时所给0-1背包问题的更优解， 从而$(y_1, y_2, \ldots, y_n)$不是所给0-1背包问题的最优解。 此为矛盾。

12.2 递归关系

设所给0-1背包问题的子问题

$$\max \sum_{k=i}^{n} v_k x_k$$
$$\begin{cases} \sum_{k=i}^{n} w_k x_k \le j \\ x_i \in \{0,1\}, i \le k \le n \end{cases}$$

的最优解为$m(i,j)$，即$m(i,j)$是背包容量为j, 可选择物品为$i, i+1, \ldots, n$时0-1背包问题的最优解。由0-1背包问题 的最优子结构性质，可以建立计算$m(i,j)$的递归式如下：

$$m(i,j) = \begin{cases} \max\{m(i+1,j), m(i+1, j-w_i)+v_i\} & j \ge w_i \\ m(i+1,j) & 0 \le j < w_i \end{cases}$$
$$m(n,j) = \begin{cases} v_n & j > w_n \\ 0 & 0 \le j < w_n \end{cases}$$

12.3 划分阶段

第一阶段： 只装入1个物体，确定各种不同载重量背包下，能够获得最大值。

第二阶段： 装入前2个物体，确定各种不同载重量背包下，能够获得最大值。

依此类推， 知道第n个阶段，最后m(n,c)便是载重量为c的背包下，装入n个物体时能够获得最大值。

12.4 确定装入

从m(n,c)的值向前倒推：$m(n,c) > m(n-1,c)$表明n物体装入背包，则问题简化为： 前n-1个物体被装入载重量为c-wn背包中。$m(n,c) \le m(n-1,c)$表明n物体未装入背包， 则问题简化为：前n-1个物体被装入载重量为c背包中。

依次类推，直道第1个物体是否被装入背包为止。可得递推关系：
若：$m(n,c) = m(n-1,c)$则 $x_i = 0$
若：$m(n,c) > m(n-1,c)$则 $x_i = 1, j = j - w_i$

12.5 例题

设0-1背包的一个实例：n = 5, c = 10, w = {2, 2, 6, 5, 4}, v = { 6, 3, 5, 4, 6}
有动态函数递推$m[][]$如下：

i/j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	9	9	9	9	9
3	0	0	6	6	9	9	9	9	11	11	14
4	0	0	6	6	9	9	9	10	11	13	14
5	0	0	6	6	9	9	12	12	15	15	<u>15</u>

若$m[i][j] = m[i-1][j] xi = 0$
若$m[i][j] > m[i-1][j] xi = 1$
故,最优解为(1, 1, 0, 0, 1)

12.6 算法实现

12.6.1 动态规划求最优值

```
void knapsack(v[], w[], c, n, x[]) {
    初始化 m[][], x[];
    for(int i = 0; i <= n; i++) {
        for(int j = 1; j <= c; j++) {
            m[i][j] = m[i-1][j];
            if(j >= w[i] && (m[i-1][j-w[i]] + v[i] > m[i-1][j])
                m[i][j] = m[i-1][j-w[i]] + v[i];
        }
    }
}
```

12.6.2 构造最优解

```
public static void traceback(m[][], w[], c, n, x[]) {
    j = c;
    for(int i = n; i < 0; i--) {
        if(m[i][j] > m[i-1][j]) {
            x[i] = 1;
            j = j - w[i];
        }
    }
}
```

12.7 算法复杂度分析

从m(i,j)的递归式容易看出,算法需要$O(nc)$计算时间.

当背包容量c很大时，算法需要的计算时间较多。例如当$c > 2n$时， 算法需要$\Omega(n^2)$计算时间。

13 n后问题

在$n \times n$的棋盘上放置彼此不受攻击的n个皇后。 按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。 n后问题等价于在$n \times n$格的棋盘上放置n个皇后， 任何2个皇后不放在同一行或同一列或同一斜线上。

13.1 算法描述

13.1.1 解向量

$(x_1, x_2, \ldots, x_n)$

13.1.2 显约束

x_i序号表示所在行，其值表示所在列。

13.1.3 隐约束

1. 不同列：$x_i \neq x_j$
2. 不处于同一正、反对角线：$|i - j| \neq |x_i - x_j|$

```
public static boolean place(int k) {
    for(int j = 1; j < k; j++)
        if((abs(k-j) == abs(x[j]-x[k])) || (x[j] == x[k]))
            return false;
    return true;
}

public static void backtrack(int t) {
    if(t > n)
        sum++;
    else {
        for(int i = 1; i <=n; i++) {
            x[t] = i;
            if(place(t))
                backtrack(t+1);
        }
    }
}
```

14 图的m着色问题

给定无向连通图G和m种不同的颜色。用这些颜色为图G的个顶点着色， 每个顶点着一种颜色。是否有一种着色法使G中每条边的2个顶点着不同的颜色。 这个问题是图的m可着色判定问题。 若一个图最少需要m种颜色才能使土中每条边连接的2个顶点着不同颜色， 则称这个数m为该图的色数。 求一个图的色数m的问题称为图的m可着色优化问题。

14.1 算法描述

14.1.1 解空间

$(x_1, x_2, \ldots, x_n)$表示顶点i所着颜色$x[i]$。

14.1.2 可行性约束函数

顶点i与已着色的相邻顶点颜色不重复。
邻接矩阵 G = (V,E)
$a[i,j]\begin{cases}0 \\ 1 \quad \text{存在边}\end{cases}$
$x[i]$ = 表示着色
n = 3, m = 3解空间树
n + 1层为叶结点

```
public static void backtrack(int t) {
    if(t > n) {
        sum++;
    } else {
        for(int i = 1; i <= m; i++) {
            x[t] = i;
            if(ok(t))
                backtrack(t+1);
        }
    }
}
```

```
public static boolean ok(int k) {
    for(int j = 1; j <= n; j++) {
        if(a[k][j] && (x[j] == x[k]))
            return false;
    }
}
```

14.2 算法复杂度

图m可着色问题的解空间树中内结点个数是：$\sum_{i=0}^{n-1} m^i$对于每一个内结点， 在最坏情况下，用ok检查当前扩展结点的每一个儿子所相应的颜色可用性需耗时$O(mn)$。 因此，回溯法总的时间耗费是：

$$\sum_{i=0}^{n-1}(mn) = \frac{nm(m^n-1)}{m-1} = O(nm^n)$$

15 最后一道设计题

15.1 思路

将N个数组分为大致相同的两半，看x是不是在$a[N/2]$这个数组的范围内。

若在，则在$a[N/2]$这个数组通过常数次比较得出结果(有返回N/2，无返回-1)

若不在，考虑两种情况：$\begin{cases} x > a[N/2]\text{中最大的数，则在}\frac{N}{2}+1-N-1\text{的数组中继续查找} \\ x < a[N/2]\text{中的最小数，则在}0-\frac{N}{2}-1\text{的数组中继续查找} \end{cases}$

```
public static int search(int a[], int n, int x) {
    for(int i = 0; i < n; i++) {
        if(a[i] == x)
            return i;
    return -1;
}

public static int divideSearch(int a[][5], int n, int x) {
    int left = 0;
    int right = n-1;
    while(left <= right) {
        int middle = (left + right) / 2;
        if((x >= a[middle][0] && x <= a[middle][4]) ||
            (x <= a[middle][0] && x >= a[middle][4]))
            return (search(a[middle], 5, x) == -1) ? -1 : middle;
        else if(x > a[middle][0] && x > a[middle][4])
            left = middle + 1;
        else
            right = middle - 1;
    }
    return -1;
}
```

15.2 算法复杂度分析

该算法的复杂度主要在于二分查找数组，所以其时间复杂度为：$T(n) = \log_2(N)$。