

算法设计专题报告

学院：信息科学与工程学院

专业：计算机应用技术

姓名：邓小龙

学号：1001690

2011 年 5 月

目 录

| | |
|-------------------------|----|
| 作业一 全排列的生成算法 | 1 |
| 作业二 算法设计基本技术 | 6 |
| 作业三 NP 完全问题的近似算法 | 16 |
| 作业四 线性规划单纯性表格实现 | 19 |
| 作业五 整数线性规划算法实现 | 23 |

作业一 全排列的生成算法

1.1 算法应用背景

该算法对于给定的字符集，能将将所有可能的全排列无重复无遗漏地枚举出来。这样就可以保证我们解决某个排列问题不会丢失任何一种排列情况。这样就与我们解决一些实际问题非常有用。如我们给定一些数字想知道这些数字的全排列是什么样的，这要是人来查是相当大的工程。所以全排列算法对我们来说相当的有用。

1.2 多种算法求解

1.2.1 回溯法

一、算法原理

回溯法(探索与回溯法)是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

二、算法描述

以 3 个元素为例；树的节点有个数据，可取值是 1、2、3。如果某个为 0，则表示尚未取值。初始状态是(0, 0, 0)，第 1 个元素值可以分别挑选 1, 2, 3，因此扩展出 3 个子结点。用相同方法找出这些结点的第 2 个元素的可能值，如此反复进行，一旦出现新结点的 3 个数全非零，那就找到了一种全排列方案。当尝试了所有可能方案，即获得了问题的解答。

三、程序实现及程序截图

代码 1.1 回溯法代码实现(recallPROG.cpp)

```
void remo(int p[],int n,int k)
{
    int b;
    if (k>n) outp(p,n);
    else
    {
        for(int i=1;i<=n;i++)
        {
            b=0;
            p[k]=i;
            for(int j=1;j<k;j++)
                if(i==p[j])
                {
                    b=true;
                    break;
                }
            if(b==0) remo(p,n,k+1);
        }
    }
}
```

```
}
```

代码 1.2 将排列按一行 5 个进行输出的实现

```
void outp(int p[],int n) //输出一个排列
{
    int i;
    cout<<" ";
    for(i=1;i<=n;i++)
    {
        cout<<p[i];
    }
    fir++;
    cout<<" ";
    if(fir%5==0) cout<<endl;
}
```

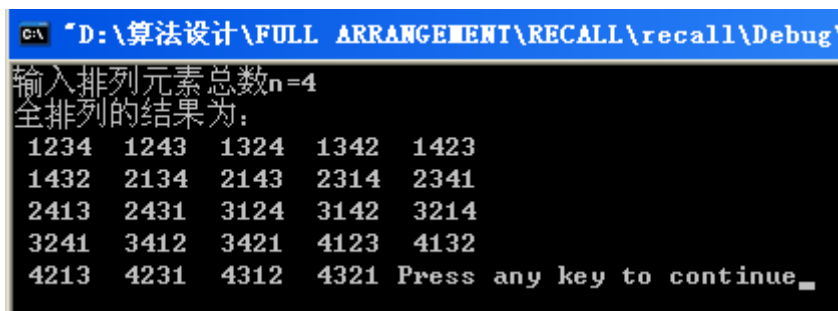


图 1.1 回溯法解决全排列的程序截图

1.2.2 字典序生成算法

一、算法原理

字典序生成算法是一种常用的全排列生成算法,它对给定的字符集中的字符规定了一个先后关系,在此基础上规定两个全排列的先后是从左到右逐个比较对应的字符的先后。字符集{1,2,3},较小的数字较先,这样按字典序生成的全排列是: {123, 132, 213, 231, 312, 321}。一个全排列可看作一个字符串,字符串可有前缀、后缀。所谓一个全排列的下一个就是这一个与下一个之间没有其他的。这就要求这一个与下一个有尽可能长的共同前缀,也即变化限制在尽可能短的后缀上。例如, 839647521 是 1-9 的排列。1-9 的排列最前面的是 123456789,最后面的是 987654321,从右向左扫描若都是增的,就到了 987654321,也就没有下一个了。否则找出第一次出现下降的位置。

二、算法描述

- 1) 设 $P_1..n$ 是 $1..n$ 的一个全排列 $P = P_1P_2 \dots P_n = P_1P_2 \dots P_{j-1}P_jP_{j+1} \dots P_{k-1}P_kP_{k+1} \dots P_n$ 从排列的右端开始,找出第一个比右边数字小的数字的序号 j (j 从左端开始计算),即 $j = \max \{ i \mid P_i < P_{i+1} \}$; 如果 j 不存在,在当前全排列为最后一个全排列,返回错误,算法结束;如找到 $j \geq 1$ 则转至 2)。
- 2) 在 P_j 的右边的数字中,找出所有比 P_j 大的数中最小的数字 P_k ,即: $k = \min \{ i \mid P_i > P_j \}$ (右边的数从右至左是递增的,因此 k 是所有大于 P_j 的数字中序号最大者)。
- 3) 将 P_i, P_k 位置对换。

- 4) 将调整后的排列字串 $P_{j+1} \dots P_{k-1}P_kP_{k+1} \dots P_n$ 的顺序进行逆序操作, 得到新的排列 $P' = P_1P_2 \dots P_{j-1}P_jP_n \dots P_{k+1}P_kP_{k-1} \dots P_{j+1}$, 此排列即为所求的下一排列。算法结束。

三、程序实现及程序截图

代码 1.3 字典序全排列生成函数(dictPROG.cpp)

```
void perm(){
    cout<<"进行排列操作的字符串为: ";
    cout<<input<<endl;
    cout<<"全排列的结果为: "<<endl;
    while(1){
        int index=-1;
        for(int a=input.size()-2;a>=0;a--){
            if(input[a]<input[a+1]){
                index=a;
                break;
            }
        }
        if(index===-1)break;
        char M='9';
        int C;
        for(int i=input.size()-1;i>=index+1;i--){
            if(input[i]<=input[index])continue;
            if(input[i]<=M){
                C=i;
                M=input[i];
            }
        }
        input[C]=input[index];
        input[index]=M;
        int len=input.size()-1-index;
        for(int w=1;w<=len/2;w++){
            char t=input[index+w];
            input[index+w]=input[input.size()-w];
            input[input.size()-w]=t;
        }
        cout<<input<<" ";
        fir++;
        if(fir%5==0)cout<<endl;
    }
}
```

```

C:\ "D:\算法设计\FULL ARRANGEMENT\LEXICOGRAPHIC\dict\De
进行排列操作的字符串为: 1234
全排列的结果为:
1243 1324 1342 1423 1432
2134 2143 2314 2341 2413
2431 3124 3142 3214 3241
3412 3421 4123 4132 4213
4231 4312 4321 Press any key to continue

```

图 1.2 字典序程序执行截图

1.2.3 递归法

一、算法原理

能采用递归描述的算法通常有这样的特征：为求解规模为 N 的问题，设法将它分解成规模较小的问题，然后从这些小问题的解方便地构造出大问题的解，并且这些规模较小的问题也能采用同样的分解和综合方法，分解成规模更小的问题，并从这些更小问题的解构造出规模较大问题的解。特别地，当规模 $N=1$ 时，能直接得解。

二、算法描述

令 $E = \{e_1, \dots, e_n\}$ 表示 n 个元素的集合，我们的目标是生成该集合的所有排列方式。令 E_i 为 E 中移去元素 i 以后所获得的集合， $\text{perm}(X)$ 表示集合 X 中元素的排列方式， $e_i.\text{perm}(X)$ 表示在 $\text{perm}(X)$ 中的每个排列方式的前面均加上 e_i 以后所得到的排列方式。例如，如果 $E = \{a, b, c\}$ ，那么 $E_1 = \{b, c\}$ ， $\text{perm}(E_1) = (b\ c, c\ b)$ ， $e_1.\text{perm}(E_1) = (a\ b\ c, a\ c\ b)$ 。

对于递归的基本部分，采用 $n=1$ 。当只有一个元素时，只可能产生一种排列方式，所以 $\text{perm}(E) = (e)$ ，其中 e 是 E 中的唯一元素。当 $n > 1$ 时， $\text{perm}(E) = e_1.\text{perm}(E_1) + e_2.\text{perm}(E_2) + e_3.\text{perm}(E_3) + \dots + e_n.\text{perm}(E_n)$ 。这种递归定义形式是采用 n 个 $\text{perm}(X)$ 来定义 $\text{perm}(E)$ ，其中每个 X 包含 $n-1$ 个元素。至此，一个完整的递归定义所需要的基本部分和递归部分都已完成。

三、程序实现及程序截图

代码 1.4 递归法解全排列问题的递归函数(abexchangePROG.cpp)

```

void Perm(char *List, int m, int k)
{
    static int count=0;
    if(m==k)
    {
        //cout<<++count<<":";
        for(int i=0; i<=ListLength-1; i++)
        {
            cout<<List[i];
        }
        //cout<<endl;
        fir++;
        cout<<" ";
        if(fir%5==0)
            cout<<endl;
    }
}

```

```
else
{
    for(int i=m; i<=k; i++)
    {
        Swap(List[m],List[i]);
        Perm(List, m+1, k);
        Swap(List[m],List[i]);
    }
}
```

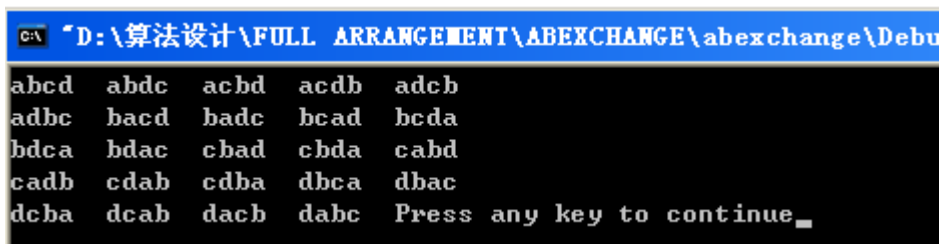
A screenshot of a Windows command prompt window. The title bar shows the path "D:\算法设计\FULL ARRANGEMENT\ABEXCHANGE\abexchange\Debu". The command prompt displays the following output:
abcd abdc acbd acdb adcb
adbcb bacd badc bcad bcda
bdca bdac cbad cbda cabd
cadb cdab cdba dbca dbac
dcba dcab dacb dabc Press any key to continue_

图 1.3 递归法的程序截图

1.3 学习或程序调试心得

通过对全排列的 3 种生成算法学习,使我了解到了如何能够求的一个数符集的所有可能的全排列无重复无遗漏地枚举出来。在调试程序的过程中,刚开始的时候当输入排序数列的过程中总会发现结果不是预期的结果也就是说总是出现一些错误,最后发现原来是自己根本没有完全理解算法,通过对算法的重新研究也就对算法有了新一步的了解,在回来编写程序,这样一次就使程序得到了正确的结果。所以在编写程序的时候一定要先对算法进行深入的研究和分析,这样就会保证程序编写的正确。

作业二 算法设计基本技术

2.1 0-1 背包问题应用背景

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 ，问应如何选择装入背包中的物品，使得装入背包中物品的总价值最大？

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部分 i 。

2.2 多种算法求解 0-1 背包问题

本次实验分别用 4 种算法思想解决了 0-1 背包问题，并分别通过编程予以实现。对于每个算法，实验中均完成了完整的输入提示和输入检错。

2.2.1 0-1 背包问题的动态规划算法实现

一、算法原理

“一个过程的最优决策具有这样的性质：即无论其初始状态和初始决策如何，其今后诸策略对以第一个决策所形成的状态作为初始状态的过程而言，必须构成最优策略”。简言之，一个最优策略的子策略，对于它的初态和终态而言也必是最优的。

这个“最优化原理”如果用数学化一点的语言来描述的话，就是：假设为了解决某一优化问题，需要依次作出 n 个决策 D_1, D_2, \dots, D_n ，如若这个决策序列是最优的，对于任何一个整数 $k, 1 < k < n$ ，不论前面 k 个决策是怎样的，以后的最优决策只取决于由前面决策所确定的当前状态，即以后的决策 $D_{k+1}, D_{k+2}, \dots, D_n$ 也是最优的。

最优化原理是动态规划的基础。任何一个问题，如果失去了这个最优化原理的支持，就不可能用动态规划方法计算。能采用动态规划求解的问题都需要满足一定的条件：

- (1) 问题中的状态必须满足最优化原理；
- (2) 问题中的状态必须满足无后效性。

所谓的无后效性是指：“下一时刻的状态只与当前状态有关，而和当前状态之前的状态无关，当前的状态是对以往决策的总结”。

二、算法描述

- 找出最优解的性质，并刻画其结构特征。
- 递归地定义最优值。
- 以自底向上的方式计算出最优值。
- 根据计算最优值时得到的信息，构造一个最优解。

其中 1)——3) 步是动态规划算法的基本步骤。在只需要求出最优值的情形，步骤 4) 可以省去。若需要求出问题的一个最优解，则必须执行步骤 4)。此时，在步骤 3) 中计算最优值时，通常需记录更多的信息，以便在步骤 4) 中，根据所记录的信息，快速构造出一个最优解。

三、程序实现及程序截图

代码 2.1 动态规划求解 0-1 背包问题的代码实现(dynamicPROG.cpp)

```
void dynamicFunc(int *w,int *vl){
```



```

    int c[n+1][m+1]; //从 1...i...item 物品中,背包剩余空间为 0<=j<=max_wgt
    的最大价值
    for (int i=0;i<=n;i++) //初始化 {
        for (int j=0;j<=m;j++) {
            c[i][j]=0;
        }
    }
    for (i=1;i<=n;i++) {
        for (int j=1;j<=m;j++) {
            if (w[i]<=j) {
                if (vl[i]+c[i-1][j-w[i]]>c[i-1][j])
                {
                    c[i][j]=vl[i]+c[i-1][j-w[i]]; //选择第 i 物品
                }
                else
                    c[i][j]=c[i-1][j]; //不选择第 i 个物品
            }
            else
                c[i][j]=c[i-1][j]; //剩余容量不够
        }
    }
    cout<<"最大值: ";
    cout<<c[n][m]<<endl; //返回最大值
    int temp_wei=m;
    int x[n+1]={0,0,0,0};
    for (i=n;i>0;i--) {
        if (c[i][temp_wei]==c[i-1][temp_wei]) //最后一个肯定是最大价值的
        {
            x[i]=0;
        }
        else
        {
            x[i]=1;
            temp_wei-=w[i];
        }
    }
    cout<<"选择装入背包的物品: ";
    for (i=0;i<=n;i++) {
        if (x[i]) {
            cout<<"第"<<i<<"件\t";
        }
    }
    cout<<endl;
}

```

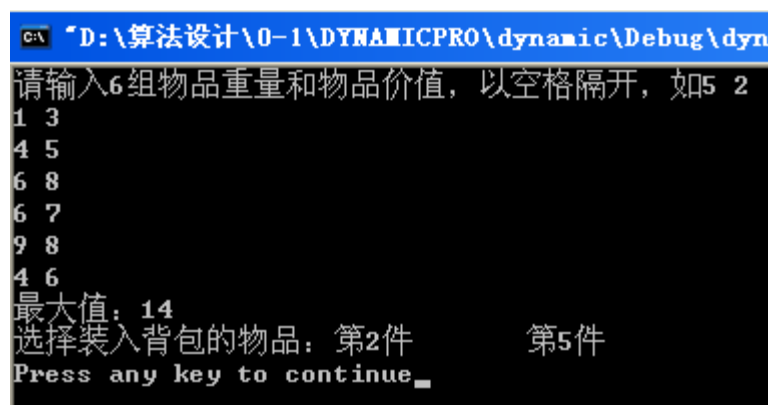


图 2.1 动态规划程序截图

2.2.2 0-1 背包问题的回溯法算法实现

一、算法原理

回溯法(探索与回溯法)是一种选优搜索法,按选优条件向前搜索,以达到目标。但当探索到某一步时,发现原先选择并不优或达不到目标,就退回一步重新选择,这种走不通就退回再走的技术为回溯法,而满足回溯条件的某个状态的点称为“回溯点”。

二、算法描述

回溯法的基本步骤:

- 针对 0-1 背包问题,定义问题的解空间;
- 确定易于搜索的解空间结构;
- 深度优先搜索解空间,并在搜索的过程中剪支。

由于回溯法是对解空间的深度优先搜索,因此可用递归函数来描述回溯法的执行过程(本实验用的是非递归实现)如下:

```
procedure try(i:integer);
var
begin
if i>n then 输出结果
else for j:=下界 to 上界 do
begin
x:=h[j];
if 可行{满足限界函数和约束条件} then begin 置值; try(i+1); end;
end;
end;
```

三、程序实现及程序截图

代码 2.2 0-1 背包问题的回溯法实现代码(recallPROGr.cpp)

```
void Knapsack(int v[6],int w[6],int c,int n,int m[6][6]){
    int jmax=(w[n]-1<c)?(w[n]-1):c;
    for(int j=0;j<jmax;j++)
        m[n][j]=0;
    for(int p=w[n];p<=c;p++)
        m[n][p]=v[n];
```

```

for(int i=n-1;i>1;i--) {
    jmax=(w[i]-1>=c)?c:(w[i]-1);
    for(int j=0;j<=jmax;j++)
        m[i][j]=m[i+1][j];

    for(int t=w[i];t<=c;t++)

m[i][t]=(m[i+1][t]>=(m[i+1][t-w[i]]+v[i]))?m[i+1][t]:(m[i+1][t-w[i]]+v[i]);
    }
    m[1][c]=m[2][c];
    if(c>=w[1])
        m[1][c]=(m[1][c]>=(m[2][c-w[1]]+v[1]))?m[1][c]:(m[2][c-w[1]]+v[1]);
}
void recallFunc(int m[6][6],int w[6],int c,int n,int x[6])
{
    for(int i=1;i<n;i++)
        if(m[i][c]==m[i+1][c])
            x[i]=0;
        else
        {
            x[i]=1;
            c-=w[i];
        }
    x[n]=(m[n][c]!=0)?1:0;
}

```

```

C:\ "D:\算法设计\0-1\RECALL\recallPROG
求解的结果是：
=====
物品的重量分别为：
0 2 2 6 5 4
物品的价值分别为：
0 6 3 5 4 6
背包的容量为： 10
=====
选择放入背包的物品是：
第1件物品
第2件物品
第5件物品
Press any key to continue

```

图 2.2 回溯法程序截图

2.2.3 0-1 背包问题的分支限界算法实现

一、算法原理

分支定界 (branch and bound) 算法是一种在问题的解空间树上搜索问题的解的方法。但

与回溯算法不同，分支定界算法采用广度优先或最小耗费优先的方法搜索解空间树，并且，在分支定界算法中，每一个活结点只有一次机会成为扩展结点。利用分支定界算法对问题的解空间树进行搜索，它的搜索策略是：

- 产生当前扩展结点的所有孩子结点；
- 在产生的孩子结点中，抛弃那些不可能产生可行解（或最优解）的结点；
- 将其余的孩子结点加入活结点表；
- 从活结点表中选择下一个活结点作为新的扩展结点。
- 如此循环，直到找到问题的可行解（最优解）或活结点表为空。分支限界法的思想是：首先确定目标值的上下界，边搜索边减掉搜索树的某些支，提高搜索效率。

二、算法描述

以宝贝问题的实例 $n=3$, $c=30$, $w=[16,15,15]$, $p=[45,25,25]$ 为例：

[A]B,C \Rightarrow B,C

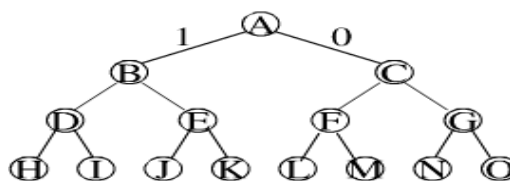
[B,C]D,E \Rightarrow E

[C,E]F,G \Rightarrow F,G

[E,F,G]J,K \Rightarrow K(45)[1,0,0]

[F,G]L,M \Rightarrow L(50)[0,1,1]M(25)

[G]N,O \Rightarrow N(25),O(0)



三、程序实现及程序截图

代码 2.3 分支限界法中用队列实现分支限界代码(branchPROG.cpp)

```
int solve(int m, int n, int p[], int w[], unsigned long* po){
    int min;
    PSeqQueue q = createEmptyQueue_seq();
    DataType x = {0,0,0,0,0,0};
    sort(n, p, w);
    x.max = up(0, m, n, p, w);
    x.min = min = down(0, m, n, p, w);
    if (min == 0) return -1;
    enqueue_seq(q, x);
    while (!isEmptyQueue_seq(q)){
        int step;
        DataType y;
        x = frontQueue_seq(q);
        dequeue_seq(q);
        if (x.max < min) continue;
        step = x.step + 1;
        if (step == n+1) continue;
        y.max = x.price + up(step, m - x.weight, n, p, w);
        if (y.max >= min) {
            y.min = x.price + down(step, m-x.weight, n, p, w);
            y.price = x.price;
            y.weight = x.weight;
            y.step = step;
            y.po = x.po << 1;
            if (y.min >= min) {
```

```

        min = y.min;
        if (step == n) *po = y.po;
    }
    enqueue_seq(q, y);
}
if (x.weight + w[step-1] <= m) {
    y.max = x.price + p[step-1] +
        up(step, m-x.weight-w[step-1], n, p, w);
    if (y.max >= min) {
        y.min = x.price + p[step-1] +
            down(step, m-x.weight-w[step-1], n, p, w);
        y.price = x.price + p[step-1];
        y.weight = x.weight + w[step-1];
        y.step = step;
        y.po = (x.po << 1) + 1;
        if (y.min >= min) {
            min = y.min;
            if (step == n) *po = y.po;
        }
        enqueue_seq(q, y);
    }
}
}
return min;
}

```

代码 2.4 队列操作代码

```

/* 在队列中插入一元素 x */
void enqueue_seq( PSeqQueue paqu, DataType x ) {
    if( (paqu->r + 1) % MAXNUM == paqu->f )
        cout<<"Full queue." <<endl;
    else {
        paqu->q[paqu->r] = x;
        paqu->r = (paqu->r + 1) % MAXNUM;
    }
}

/* 删除队列头元素 */
void dequeue_seq( PSeqQueue paqu ) {
    if( paqu->f == paqu->r )
        cout<<"Empty Queue."<<endl;
    else
        paqu->f = (paqu->f + 1) % MAXNUM;
}

```

```

/* 对非空队列,求队列头部元素 */
DataType frontQueue_seq( PSeqQueue paqu ) {
    return (paqu->q[paqu->f]);
}
/* 物品按性价比从新排序*/
void sort(int n, int p[], int w[]){
    int i, j;
    for (i = 0; i < n-1; i++)
        for (j = i; j < n-1; j++) {
            int a = p[j]/w[j];
            int b = p[j+1]/w[j+1];
            if (a < b) {
                int temp = p[j];
                p[j] = p[j+1];
                p[j+1] = temp;
                temp = w[j];
                w[j] = w[j+1];
                w[j+1] = temp;
            }
        }
}
/* 求最大可能值*/
int up(int k, int m, int n, int p[], int w[]){
    int i = k;
    int s = 0;
    while (i < n && w[i] < m) {
        m -= w[i];
        s += p[i];
        i++;
    }
    if (i < n && m > 0) {
        s += p[i] * m / w[i];
        i++;
    }
    return s;
}
/* 求最小可能值*/
int down(int k, int m, int n, int p[], int w[]){
    int i = k;
    int s = 0;
    while (i < n && w[i] <= m) {
        m -= w[i];
        s += p[i];
    }
}

```

```
        i++;
    }
    return s;
}
```

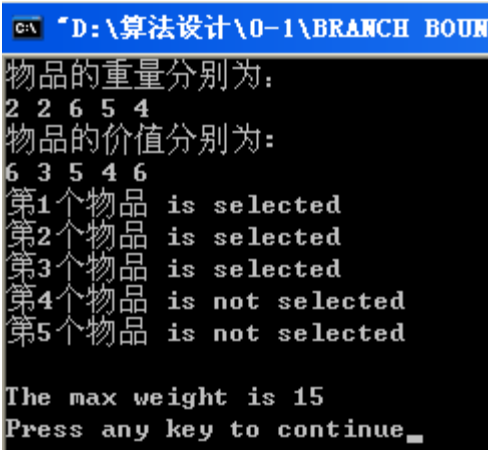


图 2.3 分支限界程序截图

2.2.4 0-1 背包问题的贪心法实现

一、算法原理

贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的仅是在某种意义上的局部最优解。贪心算法不是对所有问题都能得到整体最优解，但对范围相当广泛的许多问题他能产生整体最优解或者是整体最优解的近似解。

二、算法描述

实现该算法的过程：

- 从问题的某一初始解出发；
- while 能朝给定总目标前进一步 do
- 求出可行解的一个解元素；
- 由所有解元素组合成问题的一个可行解。

算法抽象描述为：

```
SOLUTION Greedy(a,n)
{
    SOLUTION S=Φ;
    for(i=0;i<n;i++)
    {
        x=a[i];
        if(feasible(S ∪ x))S=S ∪ x;
    }
    return s;
}
```

三、程序实现及程序截图

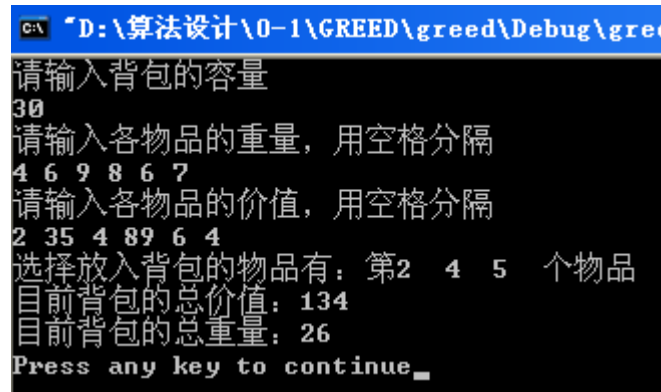
代码 2.5 效益/重量升序排列代码(greedPROG.cpp)

```
void sort(goodinfo goods[],int n) {
    int j,i;
    for(j=2;j<=n;j++) {
```

```
        goods[0]=goods[j];
        i=j-1;
        while (goods[0].rat>goods[i].rat) {
            goods[i+1]=goods[i];
            i--;
        }
        goods[i+1]=goods[0];
    }
}
```

代码 2.6 贪心选择装入背包的物品

```
void select(goodinfo goods[],int M,int n) {
    int cu;
    int i,j,value=0,weight=0;
    for(i=1;i<=n;i++)
        goods[i].X=0;
    cu=M; //背包剩余容量
    for(i=1;i<=n;i++) {
        if(goods[i].w>cu)//当该物品重量大与剩余容量跳出
            break;
        goods[i].X=1;
        value+=goods[i].p;
        weight+=goods[i].w;
        cu=cu-goods[i].w;//确定背包新的剩余容量
    }
    for(j=2;j<=n;j++) { /*按物品编号做降序排列*/
        goods[0]=goods[j];
        i=j-1;
        while (goods[0].flag<goods[i].flag)
        {
            goods[i+1]=goods[i];
            i--;
        }
        goods[i+1]=goods[0];
    }
}
```

```
C:\> "D:\算法设计\0-1\GREED\greed\Debug\greed.exe"
请输入背包的容量
30
请输入各物品的重量，用空格分隔
4 6 9 8 6 7
请输入各物品的价值，用空格分隔
2 35 4 89 6 4
选择放入背包的物品有：第2 4 5 个物品
目前背包的总价值：134
目前背包的总重量：26
Press any key to continue_
```

图 2.4 贪心算法程序截图

2.3 学习或程序调试心得

通过本次实现，我对 0-1 背包问题有了比较深刻的认识，对在本实验中用于解决背包问题的动态规划、回溯法、分支限界和贪心的实现进行了较为细致的了解。背包问题作为一种组合优化问题，它在实际生活中处处可见，只是人们没有加以细致的抽象，同时，它的多种解法的思想也都是显而易见的，难以解决的是通过程序手段实现在计算机上。

作业三 NP 完全问题的近似算法

3.1 算法应用背景

旅行售货商问题 (TSP): 给定一个无向完全图 $G=(V, E)$ 及定义在 VV 上的一个费用函数 c 和一个整数 k , 判定 G 是否存在经过 V 中各顶点恰好一次的回路, 使得该回路的费用不超过 k , 或者求得最小费用的遍历路线。

对于规模为 n 的问题, 采用近似算法可以在很短的时间内得到问题的解 (特别是与指数时间相比较)。这样对于所有的 NP 完全问题都可以使用这种算法来通过使用很短的时间就可以解决。

3.2 算法原理

首先, 算法使用的输入数据本身就是近似的; 其次, 很多问题的最优解, 允许有一定程度的近似; 算法的近似解能满足一定的精度;

误差分析基本原理为: 若一个最优化问题的最优值为 c^* , 求解该问题的一个近似算法求得的近似最优解相应的目标函数值为 c ,

则将该近似算法的性能比定义为 $\max(c/c^*, c^*/c)$ 。在通常情况下, 该性能比是问题输入规模 n 的一个函数 $\rho(n)$, 即 $\max(c/c^*, c^*/c) \leq \rho(n)$ 。

该近似算法的相对误差定义为 $\text{Abs}[(c-c^*)/c^*]$ 。若对问题的输入规模 n , 有一函数 $\varepsilon(n)$ 使得 $\text{Abs}[(c-c^*)/c^*] \leq \varepsilon(n)$, 则称 $\varepsilon(n)$ 为该近似算法的相对误差界。近似算法的性能比 $\rho(n)$ 与相对误差界 $\varepsilon(n)$ 之间显然有如下关系:

$$\varepsilon(n) \leq \rho(n) - 1。$$

3.3 算法描述

- 1) 构造图 G 的最小代价生成树 T , 遍历 T 的顶点, 把 T 转换为一条哈密尔顿回路 L 。
- 2) 用普里姆算法构造图 G 的最小代价生成树 T ;
- 3) 由最小代价生成树 T 的边集构造 T 的二叉存储数组 $tree$;
- 4) 前序遍历最小代价生成树 T , 则数组 L 中按先序遍历顺序存放的顶点序号即为问题的近似解。

3.4 程序实现及程序截图

代码 3.1 近似算法的代码实现(nearnesspPROG.cpp)

```
template<class ValueType, class ResType>
void MST_salesman_app(vector<vector<ValueType> >& C, int n, vector<int>& L,
    ResType &f, double(*func)(double)=Skip) {
    int i, idx;
    vector<pair<int, int> > mst(n);
    _prim(C, n, mst);
```

```

idx = 0;
_pretravel(mst, 0, idx, L);
for (f = func(C[L[n - 1]][L[0]]), i = 1; i < n; i++)
    f += func(C[L[i - 1]][L[i]]);
}

```

代码 3.2 二叉树的前序遍历

```

void _pretravel(vector<pair<int, int> >& tree, int cur, int&idx,
    vector<int>& path) {
    if (cur == -1)
        return;
    path[idx++] = cur;
    _pretravel(tree, tree[cur].first, idx, path);
    _pretravel(tree, tree[cur].second, idx, path);
}

```

代码 3.3 prim 算法实现

```

void _prim(vector<vector<ValueType> >& C, int n, vector<pair<int, int> >& mst) {
    vector<ValueType> low(n);
    vector<bool> sel(n);
    vector<int> closet(n);
    sel[0] = true;
    closet[0] = 0;
    fill(mst.begin(), mst.end(), pair<int, int> (-1, -1));
    int i, j, k;
    for (i = 1; i < n; i++) {
        low[i] = C[0][i];
        closet[i] = 0;
        sel[i] = false;
    }
    for (i = 1; i < n; i++) {
        bool bfirst = true;
        ValueType mv = 0;
        j = 0;
        for (k = 1; k < n; k++) {
            if ((!sel[k]) && (bfirst || low[k] < mv)) {
                mv = low[k];
                j = k;
                bfirst = false;
            }
        }
        sel[j] = true;
        if (-1 == mst[closet[j]].first) {
            mst[closet[j]].first = j;

```

```

    } else {
        int l = mst[closet[j]].first;
        while (mst[l].second != -1)
            l = mst[l].second;
        mst[l].second = j;
    }
    for (k = 1; k < n; k++) {
        if ((!sel[k]) && (C[j][k] < low[k])) {
            low[k] = C[j][k];
            closet[k] = j;
        }
    }
}
}
}

```

```

C:\ "D:\算法设计\NEARNESS\nearnessp\Debug\nearnessp.exe"
please input the count of the points:
5
please input the coordinate(坐标) of the point1
3 5
please input the coordinate(坐标) of the point2
4 6
please input the coordinate(坐标) of the point3
7 8
please input the coordinate(坐标) of the point4
2 9
please input the coordinate(坐标) of the point5
1 9
MinCost: 15.5909
Approximate TSP Tour Path as Below:
A-B-C-D-E-A
Press any key to continue

```

图 3.1 近似算法程序截图

3.5 学习或程序调试心得

对于该算法的学习与编程实现使我懂得了一个 NP 完全问题的解决步骤极其解决策略：(1)只对问题的特殊实例求解；(2)用动态规划法或分支限界法求解(3)用概率算法求解；(4)只求近似解；(5)用启发式方法求解。NP 完全问题的求解都是一个非常麻烦的事情。

在编程过程中最让我烦心的就是指针的指向到那个节点了，为了能够很好的实现该算法，我采用了栈的基本思想，而且用数组实现栈的基本功能，这样做比较方便入栈和出栈。

作业四 线性规划单纯性表格实现

4.1 算法应用背景

单纯形算法对于线性规划的问题的解决非常有帮助，例如它可以解决运输问题，生产问题等。对于任何线性规划问题的求解都是非常方便。

单纯形是美国数学家 G.B.丹齐克于 1947 年首先提出来的。原单纯形法不是很经济的算法。1953 年美国数学家 G.B.丹齐克为了改进单纯形法每次迭代中积累起来的进位误差，提出改进单纯形法。其基本步骤和单纯形法大致相同，主要区别是在逐次迭代中不再以高斯消去法为基础，而是由旧基阵的逆去直接计算新基阵的逆，再由此确定检验数。这样做可以减少迭代中的累积误差，提高计算精度，同时也减少了在计算机上的存储量。

4.2 算法原理

把线性不等式组 $AX \leq B$ 转变为线性方程组，引进松弛变量 $X_{n+1}, X_{n+2}, \dots, X_{n+m}$ ，这样就会使不等式组 $AX \leq B$ 转化为 $AX = B$ ，转换为线性方程组之后先找出一个基本可行解，对它进行鉴别，看是否是最优解；若不是，则按照一定法则转换到另一改进的基本可行解，再鉴别；若仍不是，则再转换，按此重复进行。因基本可行解的个数有限，故经有限次转换必能得出问题的最优解。

4.3 算法描述

一般单纯形法的算法：

- 1) 把线性规划问题的约束方程组表达成典范型方程组
- 2) 构造方程组的基本可行解。若基本可行解不存在，即约束条件有矛盾，则问题无解，返回错误，算法结束，否则转到 3)
- 3) 从初始基本可行解作为起点，根据最优性条件和可行性条件，引入非基变量取代某一基变量，找出目标函数值更优的另一基本可行解。
- 4) 按步骤 3) 进行迭代，直到对应检验数满足最优性条件（这时目标函数值不能再改善），即得到问题的最优解。
- 5) 若迭代过程中发现问题的目标函数值无界，则终止迭代，算法结束。

两段法的求解算法：

- 1) 根据输入数据构造 $(m+n-1) \times (m \times n)$ 维初始矩阵
- 2) 添加 $(m+n-1)$ 个人工变量，构造增广矩阵，修改目标函数系数向量，除人工变量对应系数全为 1 外其余全为 0，计算一阶段的初始解
- 3) 根据单纯形法 3) ~ 5) 求解最小值
- 4) 如果有 3) 无解或求得最小值不为 0，则模型无解，算法结束，否则转 5)
- 5) 删除人工变量及增广矩阵中的对应列，将目标函数系数向量复原为原问题的系数。（第一阶段结束，以构造出第二阶段的初始解）

- 6) 根据单纯形法 3) ~5) 求解最小值
- 7) 将 6) 中的最终解作为初始解, 根据单纯形法 3) ~5) 求解最大值
- 8) 返回最小值和最大值, 算法结束

4.4 程序实现及程序截图

代码 4.1 单纯性法代码实现(simlexPROG.cpp)

```
void main() {
    int n=3,m=3;
    int xishu[]={0,-1,3,0,-2,0};
    float Amaxti[][3]={ {3,-1,2},{-2,4,0},{-4,3,8}};
    float B[3] = {7,12,10};
    float C[3] = {-1,3,-2};
    float peita[3];
    float z = 0;
    int Bjn[3] = {2,3,5};
    int Njn[3] = {1,4,6};
    cout<<"需要规划的线性方程组为: "<<endl;
    cout<<"max z =-x2+3*x3-2*x5"<<endl;
    cout<<"x1+3*x2-x3+2*x5=7"<<endl;
    cout<<"x4-2*x2+4*x3=12"<<endl;
    cout<<"x6-4*x2+3*x3+8*x5=10"<<endl;
    int biaoji=0;
    for(int i=0;i<=2;i++)
        if(C[i]>0) biaoji=1;
    int mn = 0;
    while(biaoji==1) {
        z=0;
        int e=max(C);
        for(mn =0;mn<3;mn++) {
            if(Amaxti[mn][e]>0) break;}
        if(mn==3) {
            cout<<"方程无解"<<endl;
        }
        for(int i =0;i<3;i++)
            peita[i]=B[i]/Amaxti[i][e];
        int k=min(peita,3);
        for(i=0;i<3;i++){
            if(i!=k) {
                B[i]=B[i]-Amaxti[i][e]*B[k]/Amaxti[k][e];
            }
        }
        B[k]=B[k]/Amaxti[k][e];
        for(i=0;i<3;i++) {
```

```

        if(i!=e) {
            C[i]=C[i]-Amaxti[k][i]*C[e]/Amaxti[k][e];
        }
    }
    C[e]=-C[e]/Amaxti[k][e];
    for(i=0;i<3;i++)
        for(int j=0;j<3;j++) {
            if(i!=k&&j!=e) {
                Amaxti[i][j]=Amaxti[i][j]-Amaxti[i][e]*Amaxti[k][j]/Amaxti[k][e];
            }
        }
    for(i=0;i<3;i++) {
        if(i!=k)
            Amaxti[i][e]=-Amaxti[i][e]/Amaxti[k][e];
    }
    for(i=0;i<3;i++) {
        if(i!=e)
            Amaxti[k][i]=Amaxti[k][i]/Amaxti[k][e];
    }
    Amaxti[k][e]=1/Amaxti[k][e];
    int w=Bjin[e];
    Bjin[e]=Njin[k];
    Njin[k]=w;
    for(i=0;i<=2;i++) {
        z=z+xishu[Njin[i]-1]*B[i];
    }
    int h;
    for(h=0;h<3;h++) {
        if(C[h]>0) break;
    }
    if(h==3) biaoji=0;
}
if(mn!=3) {
    cout<<"各个变量的取值为: ";
    for(int i=0;i<=2;i++)
        cout<<" x"<<Njin[i]<<"="<<B[i];
    for(i=0;i<=2;i++)
        cout<<" x"<<Bjin[i]<<"=0";
    cout<<endl;
    cout<<"最大值 max="<<z<<endl;
}
}
}

```

代码 4.2 最大最小值求解代码

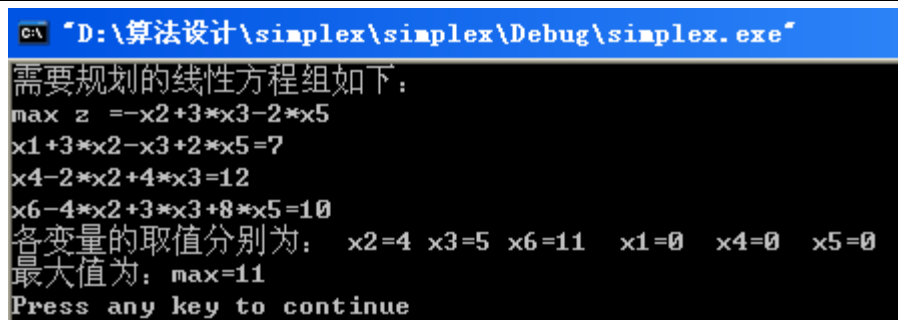
```
int min(float peita[],int n) {
```

```

float min = 1000;
int post = -1;
for(int i=0;i<n;i++){
    if(peita[i]<0) continue;
    else if(peita[i]<min) {
        min=peita[i];
        post=i;
    }
}
return post;
}

int max(float c[]) {
    float max = 0;
    int post = -1;
    for(int i=0;i<3;i++){
        if(c[i]>max) {
            max=c[i];
            post=i;
        }
    }
    return post;
}

```



```

C:\ "D:\算法设计\simplex\simplex\Debug\simplex.exe"
需要规划的线性方程组如下:
max z = -x2+3*x3-2*x5
x1+3*x2-x3+2*x5=7
x4-2*x2+4*x3=12
x6-4*x2+3*x3+8*x5=10
各变量的取值分别为: x2=4 x3=5 x6=11 x1=0 x4=0 x5=0
最大值为: max=11
Press any key to continue

```

图 4.1 单纯形法程序截图

4.5 学习或程序调试心得

通过对单纯形算法解决线性规划问题的了解，是我进一步理解了单纯性算法的解题步骤；可归纳如下：①把线性规划问题的约束方程组表达成典型型方程组，找出基本可行解作为初始基本可行解。②若基本可行解不存在，即约束条件有矛盾，则问题无解。③若基本可行解存在，从初始基本可行解作为起点，根据最优性条件和可行性条件，引入非基变量取代某一基变量，找出目标函数值更优的另一基本可行解。④按步骤 3 进行迭代，直到对应检验数满足最优性条件（这时目标函数值不能再改善），即得到问题的最优解。⑤若迭代过程中发现问题的目标函数值无界，则终止迭代。

在调试程序的过程当中发现了 Z 的最大值总是不对，总是比预计的值大很多，最后想到在每次循环开始之前要将其从新初始化。再有一个问题就是在改变 A, B, C 这几个矩阵的过程中要注意先后的顺序，要不很容易会使结果发生错误。

作业五 整数线性规划算法实现

5.1 算法应用背景

0-1 搜索算法有“通用的解题法”之称。用它可以系统的搜索一个问题的所有解或任意解，是一个即带有系统性又带有跳跃性的搜索算法。该算法对于解决装载问题、批处理作业调度、0-1 背包问题等都很有帮助。也同时可以快速的解决上述问题，更加适合解决组合数较大的问题。

5.2 算法原理

采用二叉搜索树遍历全部的可能解空间，当检测到某分枝不可能存在可行解或更优解时进行剪枝以缩短搜索时间。课件中给出了 0 值优先的搜索策略，本实验分别采用了 1 值优先和 0 值优先两种算法求解。这里只给出对于 1 值优先的描述。与 0 值优先不同，本算法每前进一步都首先将 $X_k = 0$ 压栈，尝试 $X_k = 1$ 的情况，如果不满足则将其将当前节点为根的子树剪掉，并进行回溯；当搜索到底层节点时只验证 $X_k = 1$ 的情况，不满足约束直接回溯，若满足，则先回溯到第一个 $X_k = 0$ 的节点，将其全部子树剪掉，并继续回溯搜索（因为其子树上不存在比当前解更优的可行解）；当遍历结束时，算法返回找到的最优解。

5.3 算法描述

- 1) 设 $idx:=0$ 表示当前的搜索深度， $mv:=0$ ， $path:=NULL$ 记录搜索过的最大价值及其对应的路径；设栈 S ，初始为空。
- 2) 将当前 idx 深度的物品添加到背包，计算当前路径上的重量和，如果超过最大重量转 3)，否则转 4)
- 3) $PUSH(false, S)$ ；转至 5)
- 4) $PUSH(true, S)$ ；计算当前路径的价值和，如果为搜索路径上的最大值则保存当前路径为候选路径；
- 5) 如果已到达最深节点则转 6)，否则 $idx++$ ，转 2)
- 6) $WHILE (idx >= 0 \ \&\& \ S[top] == true) \ DO \ POP(S); \ idx := idx - 1; \ END \ WHILE$
- 7) $WHILE (idx >= 0 \ \&\& \ S[top] == false) \ DO \ POP(S); \ idx := idx - 1; \ ND \ WHILE$
- 8) 若果 $idx < 0$ 转至 10)
- 9) $S[top] = false; \ idx++;$ 转至 2)
- 10) 返回 $path$ 和 mv ；算法结束

5.4 程序实现及程序截图

代码 2.7 0-1 搜索算法的代码实现(search2PROG.cpp)

```
void slove(istream&in, ostream&out) {
    int i, n;
    vector<int> V, W;
    int maxw, maxv;
```

```

vector<bool> res;
out<<"请分别输入背包的最大容量和物品数量"<<endl;
in >> maxw >> n;
if (!in.good()) {
    cerr << "Inputstream error!\n"
        "Solving process was aborted!\n";
    return;
}
out << "MaxWeight = " << maxw << "; Number = " << n << endl;
V.resize(n);
W.resize(n);
out<<"请分别输入每个物品的重量和价格"<<endl;
for (i = 0; in.good() && i < n; i++) {
    in >> W[i] >> V[i];
}
if (in.fail() || i < n) {
    cerr << "Load Input Data error!\n"
        "Solving process was aborted!\n";
    return;
}
ILP01(n, V, W, maxw, maxv, res);
out << "MaxValue: " << maxv << endl << "Resolution: ";
for (i = 0; i < n; i++)
    out << res[i] << ' ';
out << endl;
}

```

代码 2.8 左右孩子搜索代码实现

```

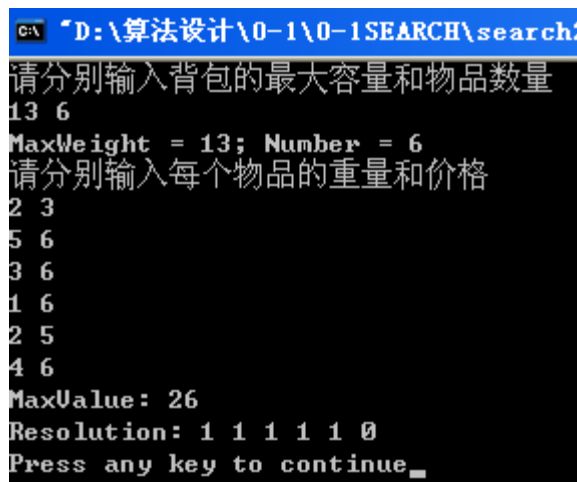
template<class VT>
void ILP01_1F(int n, vector<VT>& V, vector<VT>& W, VT maxw, VT& maxv,
vector<bool>& res) {
    stack<bool> ws;
    vector<bool> cur(n);
    res.resize(n);
    fill(res.begin(), res.end(), false);
    fill(cur.begin(), cur.end(), false);
    int idx = 0;
    VT cv = 0, cw = 0;
    maxv = 0;
    do {
        if (cw + W[idx] <= maxw) {
            ws.push(true);
            cur[idx] = true;
            cv += V[idx];

```

```

        cw += W[idx];
        if (cv > maxv) {
            copy(cur.begin(), cur.end(), res.begin());
            maxv = cv;
        }
    } else {
        ws.push(false);
    }
    if (idx == n - 1) {
        while (idx >= 0 && ws.top()) {
            ws.pop();
            cv -= V[idx];
            cw -= W[idx];
            cur[idx--] = false;
        };
        while (idx >= 0 && !ws.top()) {
            ws.pop();
            idx--;
        }
        if (idx < 0)
            return;
        ws.pop();
        cv -= V[idx];
        cw -= W[idx];
        cur[idx] = false;
        ws.push(false);
    }
    idx++;
} while (1);
}

```



```

C:\> "D:\算法设计\0-1\0-1SEARCH\search2
请分别输入背包的最大容量和物品数量
13 6
MaxWeight = 13; Number = 6
请分别输入每个物品的重量和价格
2 3
5 6
3 6
1 6
2 5
4 6
MaxValue: 26
Resolution: 1 1 1 1 1 0
Press any key to continue_

```

图 2.5 0-1 搜索法求解 0-1 背包程序截图

5.5 学习和程序调试心得

通过本次实验对回溯法有了进一步的了解，明确了回溯法的解空间，基本原理，问题的固有框架。在编程实现上没有遇到困难。回溯法的编写主要有两点，首先是要有一个界限函数，另外需要剪枝函数来提高回溯算法的效率。本实验由于要求线性方程的最大值，所以并没有适当的剪枝函数，因为，没有搜索到树底是不能知道线性方程的大小是不是最大。所以在程序的实现中，只是加了一个限制存储的控制，即，每次只保留当前线性方程和最大的组合，这样最后的即为所要求的解。