算法设计与分析 贪心算法

班级:2017211314　　学号:2017213508　　学生:蒋雪枫

一、综述

当一个问题具有最优子结构时，我们可以用动态规划算法进行求解。但当问题具有最优子结构和贪心选择性质的时候，我们考虑可以用贪心算法来进行计算，这可以简化我们求解问题的复杂度。贪心法是一个较为直接的算法，但适用的算法情景并不多，因为本身一个问题既要有最优子结构也要符合贪心选择特性，就已经挺难了。另外，贪心算法本身更重要的在于去证明其算法的正确性，这一点也会在一些例子中说明。

上次实验完成得质量不是很高，尤其是没有好好实现背包问题的求解，**同时也仅仅完成了三角剖分，但却完成得并不细致。**在本次实验中，我会结合动态规划算法和贪心法的具体实例，来对整个知识进行总结与实践，**同时补充上次写得不是很好的背包问题**。

二、再谈最优三角剖分

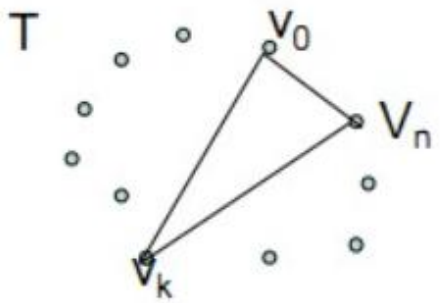
基于最优子结构的动态规划方法：

给定凸多边形 P(n+1 个点，n+1 个边)，以及定义在由多边形的边和弦组成的三角形上的权函数 w。要求确定该凸多边形的三角剖分，使得该三角剖分中诸三角形上权之和为最小。这里，**我们的目标是求解每个划分出来的三角形集合的权值之和最优解，而不是只计算边和弦。**

为什么可以用 DP 算法呢？是因为这个问题具有最优子结构。最优子结构指的是，问题的最优解包含子问题的最优解。反过来说就是，我们可以通过子问题的最优解，推导出问题的最优解。因为子问题可能被重复计算到，所以我们会记录子问题，避免重复计算。

在三角剖分算法里面，该问题具有最优子结构：

若凸(n+1)边形 P={V0,V1……Vn}的最优三角剖分 T 包含三角形 V0VkVn,1<=k<=n，则 T 的权为三个部分权之和：三角形 V0VkVn 的权，多边形{V0,V1……Vk}的权和多边形{Vk,Vk+1……Vn}的权之和。如下图所示：



一般来说，我们对于该问题的结果是**数值上唯一**的。可以断言，由 T 确定的这两个子多边形的三角剖分也是最优的。因为若有{V0,V1……Vk}和{V0,V1……Vk}更小权的三角剖分，将导致 T 不是最优三角剖分的矛盾。因此，凸多边形的三角剖分问题具有最优子结构性质。

最优三角剖分的递归结构

n+1边，整体求解目标：t[i][n]

t[i][j]代表凸子多边形{Vi-1, Vi...Vj}的最优三角剖分所对应的权函数值

•t[i][j]的值利用最优子结构性质递归地计算：

1) 当j-i≥1时，凸子多边形P{v_{i-1},v_i,…,v_j}至少有j-i+2 ≥ 3个顶点

2) 选取最优顶点v_k，，用顶点v_k，进行子问题划分

分割点在i和j之间产生

3）t[i][j]的值为三部分相加：

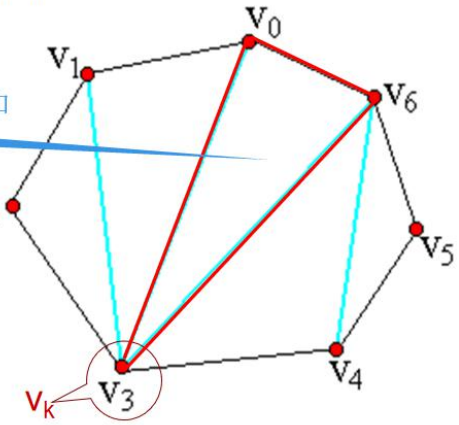
t[i][k] + t[k+1][j] + 三角形$v_{i-1}v_kv_j$权值

其中i≤k≤j-1

E.g. i=1, j=6, k=3

t[1,6]= t[1,3] + t[4,6] + w[v_0,v_3,v_6]

三边之和



t[i][j]的值可以利用最优子结构递归计算，t[i][i]=0

$$t[i][j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{t[i][k]+t[k+1][j]+w(v_{i-1}v_kv_j)\} & i<j\end{cases}$$

最后的伪代码可以如下描述，我们先一层层计算子问题(第一次循环是从 r=2 开始到 n 结束得到最后的答案)，再向右上角移动(内层第二次循环是确定开始位置，从而得到结束位置)，得到问题最后的解，这个问题和矩阵最佳连乘顺序非常相似：

```
void MinWeightTriangulation(int n, Type ** t, int **s)
{
   for (int  i = 1; i <= n; i++)   t[i][i] = 0;
   for (int  r = 2; r <= n; r++)
      for (int i = 1; i <= n-r+1; i++)
      {
         int j= i + r – 1
         t[i][j] = t[i+1][j] + w(i-1, i, j)
         s[i][j] =i;
         for (int k =i+1; k <i+r-1; k++)
          {
            int u=t[i][k] + t[k+1][j] + W(i-1,k,j);
            if (u<t[i][j])
            {
               t[i][j]) =u;
               s[i][j]) =k;
```

```
        }
      }
    }
}
```

基于贪心策略方法：

动态规划在很多情况下都能得到理想的解，但往往开销在 O（N3）这个层次，为了提高时间，我们可以试着牺牲准确度，来得到一个较好的解。因为最后一次循环，是在 i 和 j 之间找到一个最佳的分割点。如果我们限制最后一次循环的次数，就可以减小时间开销。

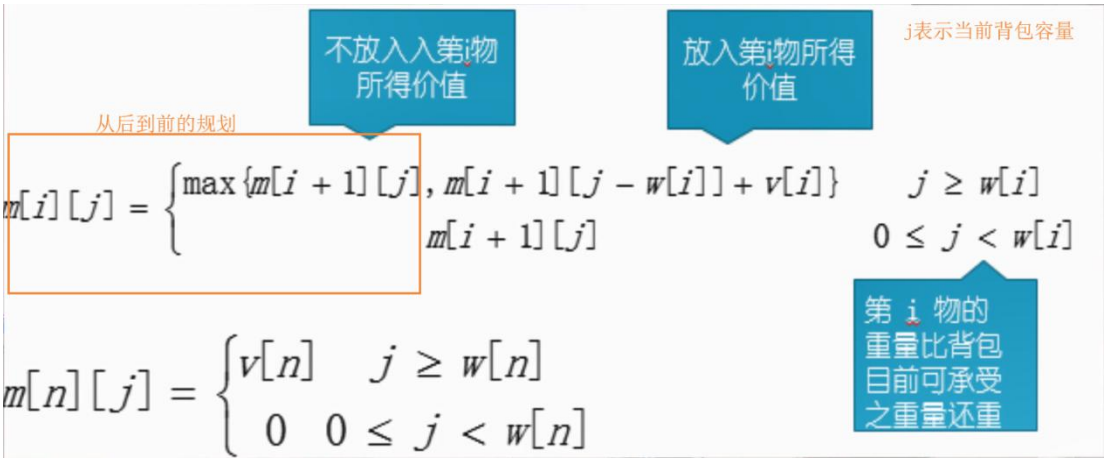
```
for (int k =i+1; k <i+r-1; k++)//把这一行换成 int k=i+1;k<i+6;k++
      {
        int u=t[i][k] + t[k+1][j] + W(i-1,k,j);
        if (u<t[i][j])
        {
          t[i][j]) =u;
          s[i][j]) =k;
        }
      }
```

但肯定算法不是精确的算法，而且随着数据量的增大，差异会愈发明显。

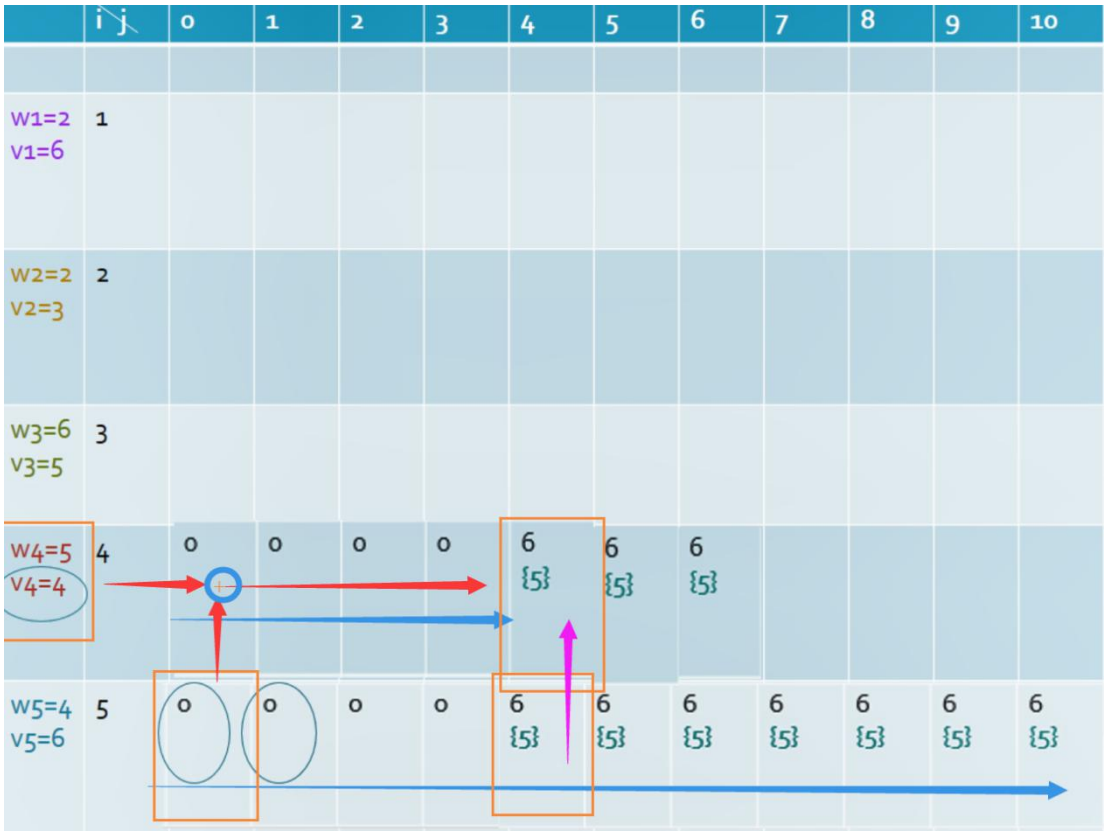
这个其实是上次的作业了：）

三、再谈背包问题

背包问题一直是一个非常常见的话题，变化多端，衍生出了许多问题，比如多重背包，完全背包等，更有诸多网友，在各种所谓的技术博客、知乎也是数见不鲜。动态规划是一种很 nice 的算法，但往往我们只是大概知道其原理,却很难有一个直观的认识。在教材中的 0-1 背包问题是一种典型的动态规划问题情景，但光看代码却很难把其中的精髓理解清晰，这里以填表的方式来说明一下背包问题的工作流程。

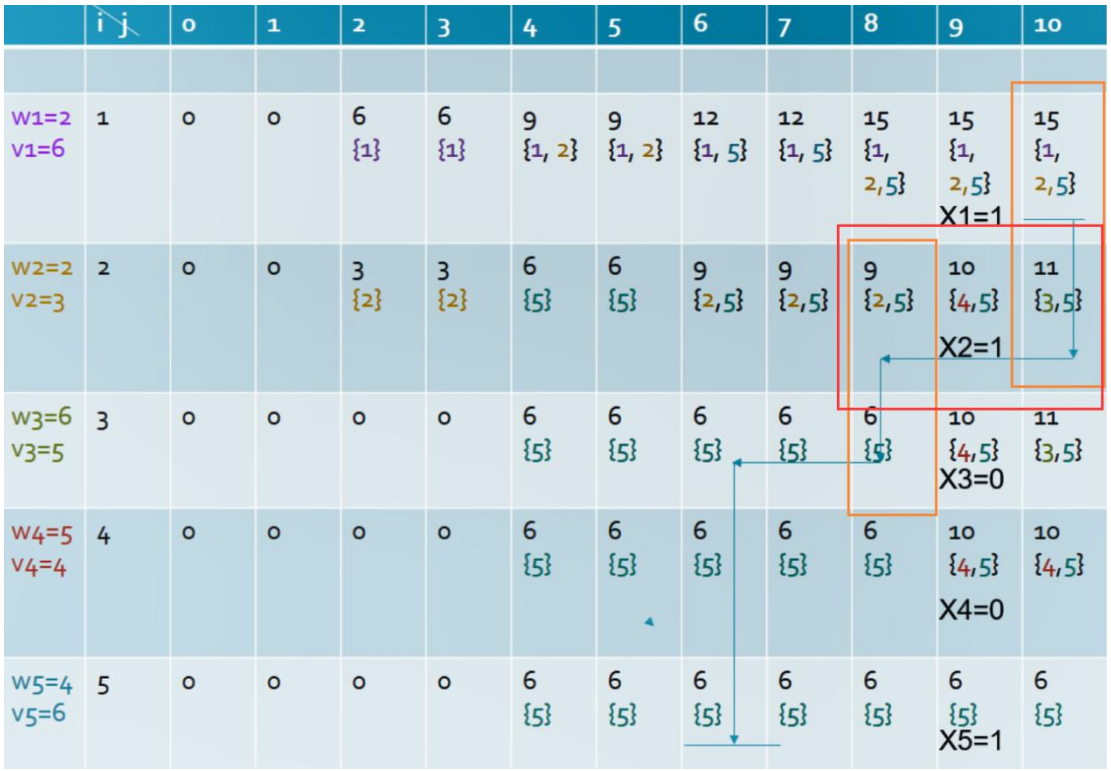


以上是 0-1 背包的状态转移方程。



这是填表的一个局部动态选择。

这是最后依次填出的表格，我们取得 M[1,10]作为最后的结果，这是就最大的价值。然后我们通过回溯，去求解向量。



源码可表示如下，还是比较直观的：

```
void Knapsack(int *v,int *w,int c,int n,int **m)
{
   int j;//current package volumn
   int jMax;
   if(w[n]-1>c) jMax=c;
   else jMax=w[n]-1;
   //末行初始化
   for(int j=0;j<=jMax;j++) m[n][j]=0;
   for(int j=w[n];j<=c;j++) m[n][j]=v[n];
   for(int i=n-1;i>1;i--)
   {
      if(w[i]-1>c) jMax=c;else jMax=w[n]-1;
      for(j=0;j<=jMax;j++)
         m[i][j]=m[i+1][j];
      for(j=w[i];j<=c;j++)
         if(m[i+1][j]>m[i+1][j-w[i]]+v[i]) m[i][j]=m[i+1][j];
         else m[i][j]=m[i+1][j-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 BackTrack(int **m,int *w,int c,int n,int x[])
{
```

```
    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]?1:0;
}
```

四、Huffman 树构造

通过课程里的学习，我们知道了哈夫曼树构造其实是贪心算法的一种体现。在合并两个节点的时候，我们选择当前频率最小的两个节点，他们会具有同样的编码长度，只是最后一位不同（若两者都为叶子），这就是贪心选择特性。而合并森林的时候，前后子树变化的部分，具有 NodeValue(T)=NodeValue(T')+f(x)+f(y)，这一点可以通过数学等式推导来证明，这属于其最优子结构特性。所以哈夫曼树一定能构造出权值和最小的一颗编码树，编码不一定唯一，但各自编码的位数和频率是一定的。

这里学生使用课本的方法和 C++ STL 的方法来完成本次任务——字符统计与编码。这里我们统一大小写，并且把标点转换为#。

源代码：

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdlib.h>
#include <string>
#include <map>
#include <fstream>
#include <sstream>
#include <iostream>
using namespace std;

//Huffman 树的节点类
typedef struct Node
{
    char value;             //结点的字符值
    int weight;             //结点字符出现的频度
    Node *lchild,*rchild;    //结点的左右孩子
}Node;
//自定义排序规则，即以 vector 中 node 结点 weight 值升序排序
bool ComNode(Node *p,Node *q)
{
    return p->weight<q->weight;
}
```

```
//构造 Huffman 树，返回根结点指针
Node* BuildHuffmanTree(vector<Node*> vctNode)
{
    while(vctNode.size()>1)                    //vctNode 森林中树个数大于 1 时循环进行合并
    {
        sort(vctNode.begin(),vctNode.end(),ComNode);  //依频度高低对森林中的树进行升序排序
        Node *first=vctNode[0];   //取排完序后 vctNode 森林中频度最小的树根
        Node *second=vctNode[1];  //取排完序后 vctNode 森林中频度第二小的树根
        Node *merge=new Node;     //合并上面两个树
        merge->weight=first->weight+second->weight;
        merge->lchild=first;
        merge->rchild=second;
        vector<Node*>::iterator iter;
        iter=vctNode.erase(vctNode.begin(),vctNode.begin()+2);   //从 vctNode 森林中删除上诉频度最小的两个节点 first
和 second
        vctNode.push_back(merge);                  //向 vctNode 森林中添加合并后的 merge 树
    }
    return vctNode[0];          //返回构造好的根节点
}
//用回溯法来打印编码
void PrintHuffman(Node *node,vector<int> vctchar)
{
    if(node->lchild==NULL && node->rchild==NULL)
    {//若走到叶子节点，则迭代打印 vctchar 中存的编码
        cout<<node->value<<": ";
        for(vector<int>::iterator iter=vctchar.begin();iter!=vctchar.end();iter++)
            cout<<*iter;
        cout<<endl;
        return;
    }
    else
    {
        vctchar.push_back(1);    //遇到左子树时给 vctchar 中加一个 1
        PrintHuffman(node->lchild,vctchar);
        vctchar.pop_back();      //回溯，删除刚刚加进去的 1
        vctchar.push_back(0);    //遇到左子树时给 vctchar 中加一个 0
        PrintHuffman(node->rchild,vctchar);
        vctchar.pop_back();      //回溯，删除刚刚加进去的 0
    }
}
string readFileIntoString(char *filename)
{
    ifstream ifile(filename);
```

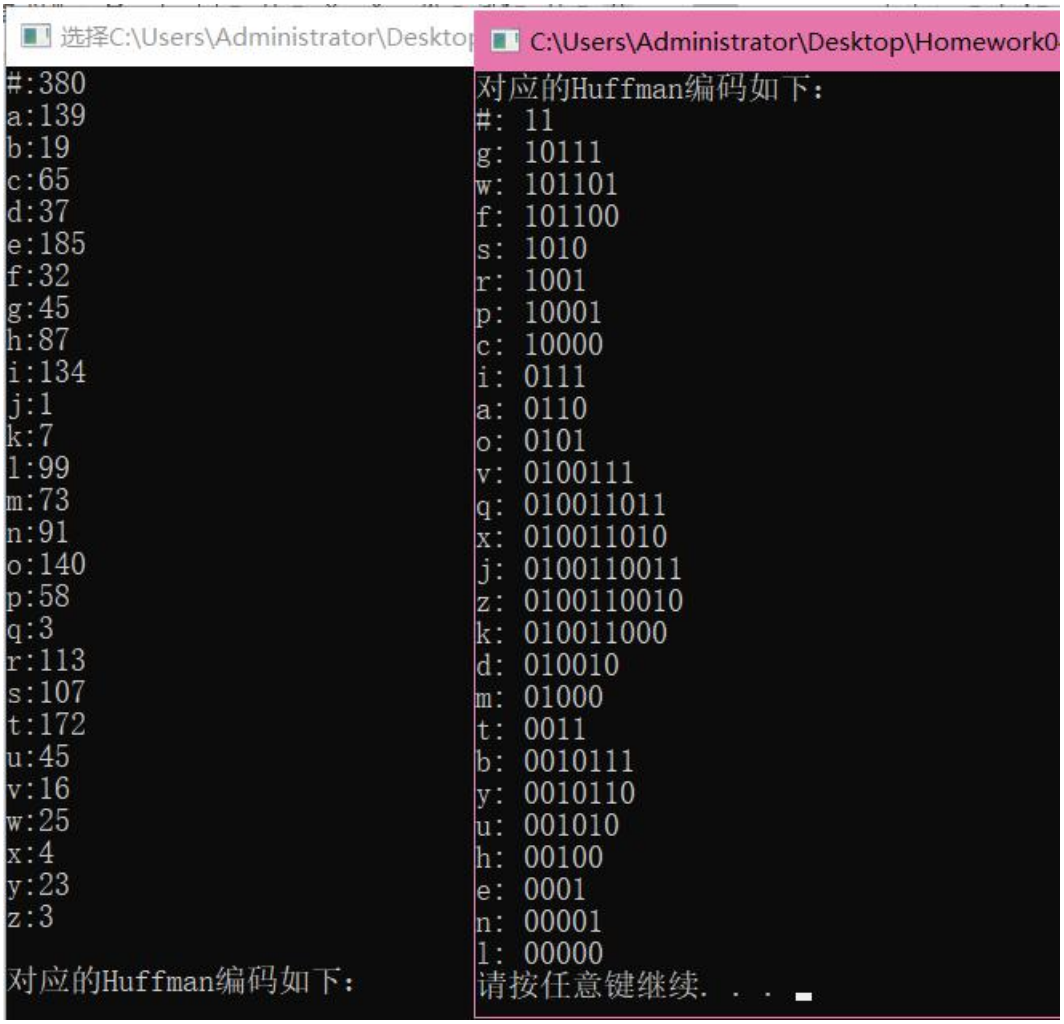
```
    //将文件读入到 ostringstream 对象 buf 中
    stringstream buf;
    char ch;
    while(buf&&ifile.get(ch))
    {
        if(isalpha(ch))
            buf.put(tolower(ch));
        else
            buf.put('#');
    }
    //返回与流对象 buf 关联的字符串
    return buf.str();
}

int main(void){

    char *fn="a.txt";
    string str;
    str=readFileIntoString(fn);
    cout<<str<<endl;

    char chars[27]={};
    int freqs[27]={};
    map<char,int> ms;
    map<char,int>::iterator p,mEnd;
    string s=str;
    int len = s.length();
    for(int i=0;i<len;i++){
        p=ms.find(s[i]);
        if(p!=ms.end()){
            p->second++;
        }
        else{
            ms.insert(pair<char,int>(s[i],1));
        }
    }
    p=ms.begin();
    mEnd=ms.end();
    int counter=0;
    for(;p!=mEnd;p++){
        cout<<p->first<<":"<<p->second<<endl;
        chars[counter]=p->first;
        freqs[counter]=p->second;
        counter+=1;
```

```
    }

    vector<Node*> vctNode;       //存放 Node 结点的 vector 容器 vctNode
    char ch;                     //临时存放控制台输入的字符
    for(int i=0;i<27;i++)
    {
        ch=chars[i];
        Node *temp=new Node;
        temp->value=ch;
        temp->lchild=temp->rchild = NULL;
        vctNode.push_back(temp);  //将新的节点插入到容器 vctNode 中
    }
    for(int i=0;i<vctNode.size();i++)
        vctNode[i]->weight=freqs[i];
    Node *root = BuildHuffmanTree(vctNode);   //构造 Huffman 树，将返回的树根赋给 root
    vector<int> vctchar;
    cout<<endl<<"对应的 Huffman 编码如下："<<endl;
    PrintHuffman(root,vctchar);
    system("pause");
}
```

运行结果：

informally##an#algorithm#is#any#welldefined#computational#procedure#that#takes#some#value##or#set#of#values##as#input#and#produces#some#value##or#set#of#values##as#output##an#algorithm#is#thus#a#sequence#of#computational#steps#that#transform#the#input#into#the#output##we#can#also#view#an#algorithm#as#a#tool#for#solving#a#well#specified#computational#problem##the#statement#of#the#problem#specifies#in#general#terms#the#desired#input#output#relationship##the#algorithm#describes#a#specific#computational#procedure#for#achieving#that#input#output#relationship##an#algorithm#is#said#to#be#correct#if##for#every#input#instance##it#halts#with#the#correct#output##we#say#that#a#correct#algorithm#solves#the#given#computational#problem##an#incorrect#algorithm#might#not#halt#at#all#on#some#input#instances##or#it#might#halt#with#an#answer#other#than#the#desired#one##contrary#to#what#one#might#expect##incorrect#algorithms#can#sometimes#be#useful##if#their#error#rate#can#be#controlled##we#shall#see#an#example#of#this#in#chapter#when#we#study#algorithms#for#finding#large#prime#numbers##ordinarily##however##we#shall#be#concerned#only#with#correct#algorithms###algorithms#for#optimization#problems#typically#go#through#a#sequence#of#steps##with#a#set#of#choices#at#each#step##for#many#optimization#problems##using#dynamic#programming#to#determine#the#best#choices#is#overkill##simpler##more#efficient#algorithms#will#do##a#greedy#algorithm#always#makes#the#choice#that#looks#best#at#the#moment##that#is##it#makes#a#locally#optimal#choice#in#the#hope#that#this#choice#will#lead#to#a#globally#optimal#solution##this#chapter#explores#optimization#problems#that#are#solvable#by#greedy#algorithms###the#greedy#method#is#quite#powerful#and#works#well#for#a#wide#range#of#problems##later#chapters#will

#present#many#algorithms#that#can#be#viewed#as#applications#of#the#greedy#method##including#minimum#spanning#tree#algorithms###dijkstra#s#algorithm#for#shortest#paths#from#a#single#source##and#chv##atal#s#greedy#set#covering#heuristic##minimum#spanning#tree#algorithms#are#a#classic#example#of#the#greedy#method##



对应相乘即可算出将这段话转化为 01 编码后的位数了。

如果有空，下面应该是用教材算法的 Huffman 树。

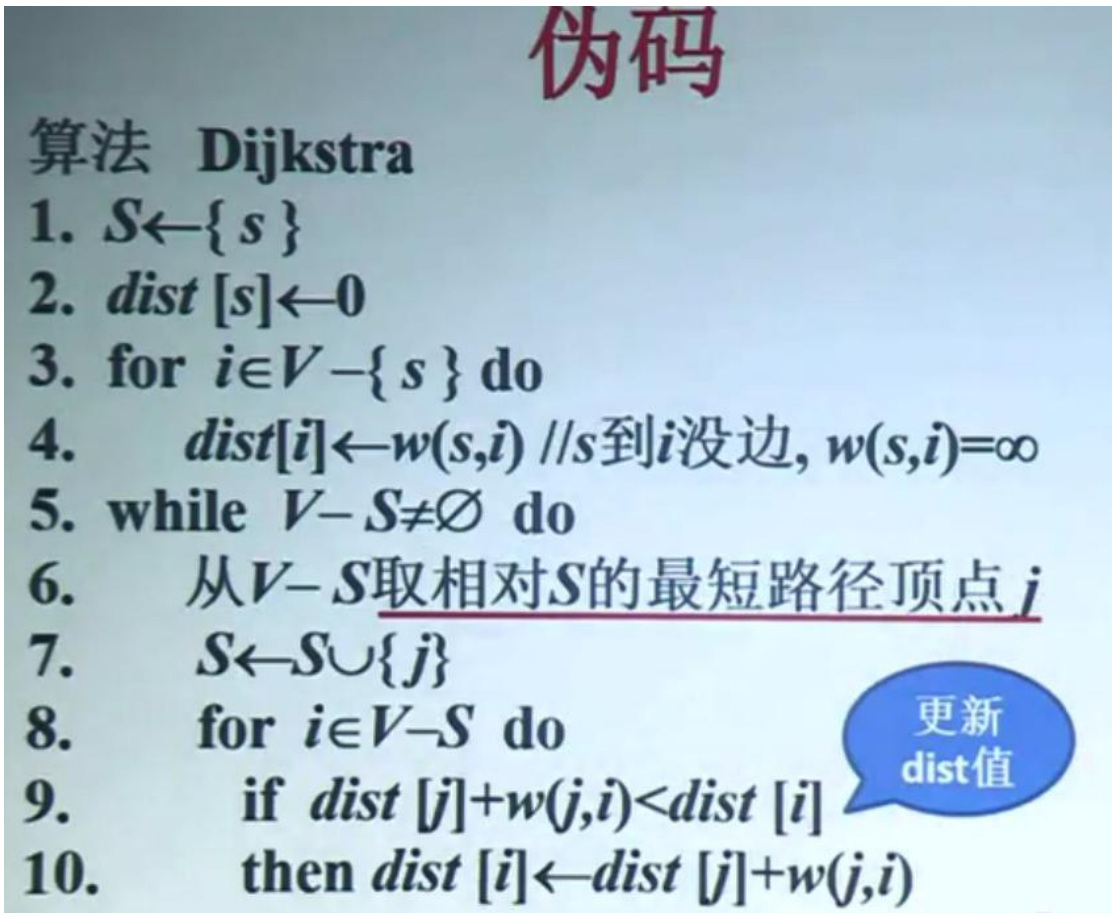
```
template<class Type>
class Huffman
{
        friend BinaryTree<int> HuffmanTree(Type [],int);
    public:
        operator Type ()const {return weight;}
    private:
        BinaryTree<int> tree;
        Type weight;
};

template <class Type>
BinaryTree<int> HuffmanTree(Type f[],int n)
```

```
{
    Huffman<Type> *w = new Huffman<Type> [n+1];
    BinaryTree<int> z,zero;
    for(int i=1;i<=n;i++)
    {
        z.MakeTree(i,zero,zero);
        w[i].weight=f[i];
        w[i].tree=z;
    }
    //bulid priority queue
    MinHeap<Huffman<Type>>Q(1);
    Q.Initialize(w,n,n);
    //Merge
    Huffman<Type> x,y;
    for(int i=1;i<n;i++)
    {
        Q.DeleteMin(x);
        Q.DeleteMin(y);
        z.MakeTree(0,x.tree,y.tree);
        x.weight+=y.weight;
        x.tree=z;
        Q.Insert(x);
    }
    Q.DeleteMin(x);
    Q.Deactivate();
    delete []w;
    return x.tree;
}
```

个人认为还可以手动模拟出 Huffman 树来实现编码的任务，应该也不是很难。

五、单源最短路

从大一，到大三，可能见过最多的算法就是 Dijsktra 算法了。通过本章的学习，我们更加深刻地学习到了该算法的正确性证明，这里我们先把算法本身展示，再回过头来证明该算法的正确性。



以上是 Dijsktra 算法的 Pseudo Code 描述，首先我们将点集分成 S 集合喝 V-S 集合，其中 V 表示全体顶点，s 表示源点 source。将 s 加入到最后的结果 dist 数组中，因为 dist[s]=0 是显然的。然后我们对其他点展开遍历，将 source 到其他点的距离先初始化，填入 dist[]数组中。初始化是根据已有的距离矩阵 W 给出。然后我们一直处理，直到 S 集合被全部填满，表示全部到达，且已经求出最短路径。先找出 V-S 集合中相对源点最短路径的定点 j，把它加入 S 集合。由于 j 点加入了 S 集合，dist 信息可能会被更新，产生最新的最短路长度（当然也可能产生更长的），如果小则更新。Dijsktra 算法的流程大抵如此。

然后我们的程序如下：

```
#include<iostream>
#include<fstream>
#include<cmath>
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<memory>
using namespace std;
double a[22][22];//only use 1-22
bool s[22];
const int MAX=1000000;
double dist[22];

void Dij(int n,int v,double *dist,int *prev,double a[][22])
```

```
{
    bool s[30];
    for(int i=0;i<=n;i++)
    {
        dist[i]=a[v][i];
        s[i]=false;
        if(dist[i]==MAX)
        {
            prev[i]=0;
        }
        else
            prev[i]=v;
    }
    dist[v]=0;s[v]=true;
    for(int i=0;i<n;i++)
        {
            int temp=MAX;
            int u=v;
            for(int j=0;j<=n;j++)
            {
                if((!s[j])&&(dist[j]<temp))
                {
                    u=j;
                    temp=dist[j];
                }
            }
            s[u]=true;
            for(int j=0;j<=n;j++)
            {
                if((!s[j])&&(a[u][j]<MAX))
                {
                    double newdist=dist[u]+a[u][j];
                    if(newdist<dist[j])
                    {
                        dist[j]=newdist;
                        prev[j]=u;
                    }
                }
            }
        }
}
int main()
{
    cout<<"Our goal is for POINT20 to find SP towards POINT1"<<endl;
```

```
    ifstream in("Matrix22.txt",ios::in);
    int i=0;
    for(int in=0;in<=21;in++)
        dist[in]=MAX;
//  printf("dist[in]:%lf",dist[1]);
    while(!in.eof())
    {
        in>>a[i][0]>>a[i][1]>>a[i][2]>>a[i][3]>>a[i][4]>>a[i][5]>>a[i][6]>>a[i][7]>>a[i][8]>>a[i][9]>>a[i][10]>>a[i][11]>>a[i]
[12]>>a[i][13]>>a[i][14]>>a[i][15]>>a[i][16]>>a[i][17]>>a[i][18]>>a[i][19]>>a[i][20]>>a[i][21];
        printf("i:%d 1:%lf 2:%lf\n",i,a[i][1],a[i][22]);
        i+=1;
    }
    int prev[22];
    Dij(21,19,dist,prev,a);
    for(int in=0;in<=21;in++)
        cout<<dist[in]<<endl;
    cout<<"target: "<<dist[0];
}
```

运行结果如下：

读取最初的权值表

```
C:\Users\Administrator\Desktop\Homework04\Dijsktra.exe
Our goal is for POINT20 to find SP towards POINT1
i:0 1:1000000.000000 2:0.000000
i:1 1:1000000.000000 2:0.000000
i:2 1:582.322915 2:0.000000
i:3 1:1000000.000000 2:0.000000
i:4 1:1000000.000000 2:0.000000
i:5 1:1000000.000000 2:0.000000
i:6 1:1000000.000000 2:0.000000
i:7 1:1000000.000000 2:0.000000
i:8 1:1000000.000000 2:0.000000
i:9 1:1000000.000000 2:0.000000
i:10 1:1000000.000000 2:0.000000
i:11 1:406.566735 2:0.000000
i:12 1:354.777897 2:0.000000
i:13 1:1000000.000000 2:0.000000
i:14 1:248.900606 2:0.000000
i:15 1:612.249343 2:0.000000
i:16 1:1000000.000000 2:0.000000
i:17 1:1000000.000000 2:0.000000
i:18 1:1000000.000000 2:0.000000
i:19 1:1000000.000000 2:0.000000
i:20 1:1000000.000000 2:0.000000
i:21 1:1000000.000000 2:0.000000
```

单源到各个点的距离：

```
1956.93
1343.41
761.938
2111.29
302.54
1988.14
683.088
1622.91
344.546
1778.06
963.852
1562.25
988.629
2072.92
1592.31
780.892
244.053
1582.91
1309.05
0
1733
810.555
target: 1956.93请按任意键继续. . .
```

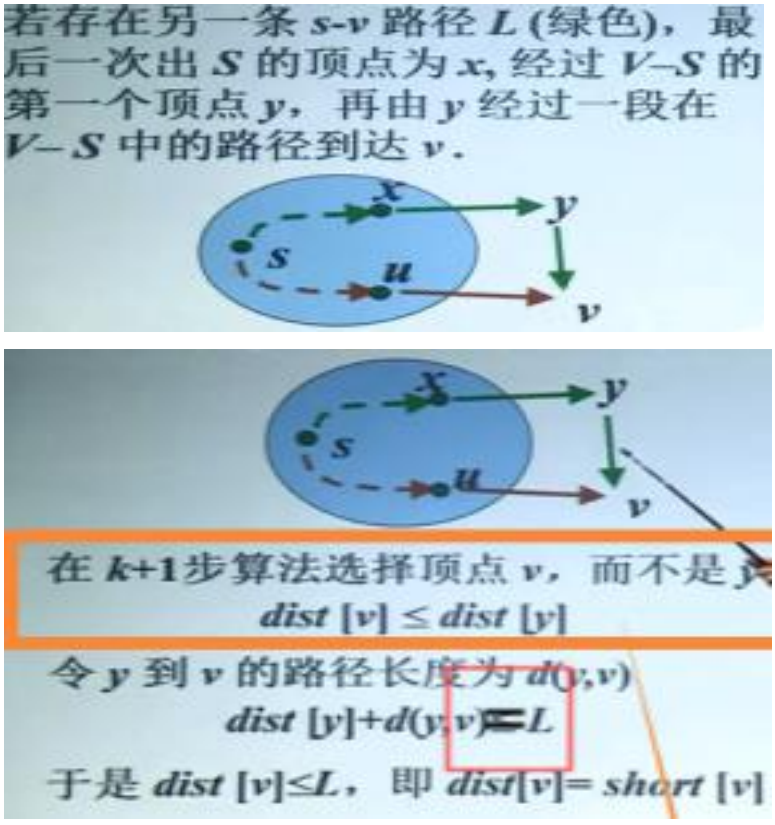
然后我们来考虑一下算法的正确性。

命题：当算法进行到第 k 步的时候，对于 S 中每一个节点，都有 dist[i]为最终的最短路长度。

归纳基础：k=1 的时候，S 仅有 source 点，dist[s]=0 为最短路径是平凡的。

归纳步骤：

假设命题对 k 来说为真，考虑 k+1 步也是正确的。我们假设这一步他选择的是顶点 v，边为<u,v>我们的目标就是证明 dist[v]也是最短的。这里，L 是指我们抽象出来的真正的最短路。



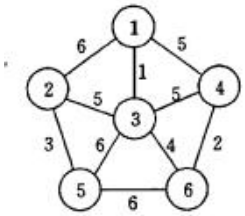
六、最小生成树 Minimum Spanning Tree

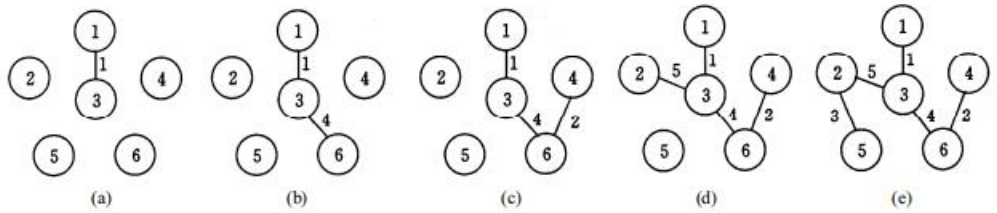
最小生成树也是贪心算法的一个实例，我们学习了 Kruscal 算法和 Prim 算法。虽然这两种算法都比较直观，但其实也不是那么好写出来的，比如我们应该如何去表示联通分支等。这里我们使用 Prim 算法来解决本次问题。

我们在这里先介绍一下 Prim 算法的工作流：

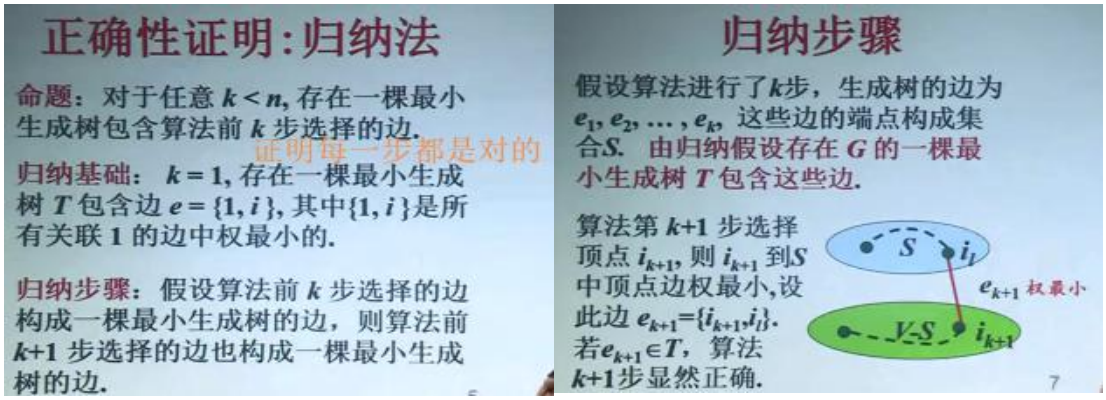
设 $G=(V, E)$是连通带权图，$V=\{1, 2, \cdots, n\}$。构造 G 的最小生成树的 Prim 算法的基本思想是：首先置 $S=\{1\}$，然后，只要 S 是 V 的真子集，就做如下贪心选择：选取满足条件 $i\in S$，$j\in V-S$，且 $c[i][j]$最小的边，并将顶点 j 添加到 S 中。这个过程一直进行到 $S=V$ 时为止，选取到的所有边恰好构成 G 的一棵最小生成树。算法描述如下：

算法结束时，T 中包含 G 的 $n-1$ 条边。利用最小生成树性质和数学归纳法容易证明，上述算法中的边集合 T 始终包含 G 的某棵最小生成树中的边。因此，在算法结束时，T 中的所有边构成 G 的一棵最小生成树。例如，对于图 4-9 中的带权图，按 Prim 算法选取边的过程如图 4-10 所示。





Prim 算法也是贪心算法的一个实例，其证明也很有趣，主要是基于最小生成树的 MST 性质和数学归纳法来证明，而 Kruscal 算法的证明较为复杂，都在教材上有所展示，这里简单说说。先把源程序展示出来。




```
#include<iostream>
#include<fstream>
#include<cmath>
#include<algorithm>
using namespace std;
#define M_PI 3.14159265358979323846
const double Earth_R=6378.137;
double a[22][22];
```

```
const int MAX=100000;


void Prim(int n,double c[][22])
{
    double lowcost[1000];
    int closest[1000];
    bool s[1000];
    s[0]=true;
    for(int i=1;i<=n;i++)
    {
        lowcost[i]=c[0][i];
        closest[i]=0;
        s[i]=false;
    }
    for(int i=0;i<n;i++)
    {
        double min=MAX;
        int j=0;
        for(int k=1;k<=n;k++)
        {
            if((lowcost[k]<min)&&(!s[k]))
            {
                min=lowcost[k];
                j=k;
            }
            cout<<"j and closest[j]:"<<j<<" "<<closest[j]<<endl;
            s[j]=true;
            for(int k=1;k<=n;k++)
            {
                if((c[j][k]<lowcost[k])&&(!s[k]))
                {
                    lowcost[k]=c[j][k];
                    closest[k]=j;
                }
            }
        }
    }
 }
int main(void)
{
    ifstream in("Matrix22.txt",ios::in);
    int i=0;
    while(!in.eof())
    {
```

```
        in>>a[i][0]>>a[i][1]>>a[i][2]>>a[i][3]>>a[i][4]>>a[i][5]>>a[i][6]>>a[i][7]>>a[i][8]>>a
[i][9]>>a[i][10]>>a[i][11]>>a[i][12]>>a[i][13]>>a[i][14]>>a[i][15]>>a[i][16]>>a[i][17]>>a[i][1
8]>>a[i][19]>>a[i][20]>>a[i][21];
        printf("i:%d 1:%lf 2:%lf\n",i,a[i][1],a[i][22]);
        i+=1;
    }
    Prim(21,a);
    system("pause");
 }
```

i:0 1:1000000.000000 2:0.000000
i:1 1:1000000.000000 2:0.000000
i:2 1:582.322915 2:0.000000
i:3 1:1000000.000000 2:0.000000
i:4 1:1000000.000000 2:0.000000
i:5 1:1000000.000000 2:0.000000
i:6 1:1000000.000000 2:0.000000
i:7 1:1000000.000000 2:0.000000
i:8 1:1000000.000000 2:0.000000
i:9 1:1000000.000000 2:0.000000
i:10 1:1000000.000000 2:0.000000
i:11 1:406.566735 2:0.000000
i:12 1:354.777897 2:0.000000
i:13 1:1000000.000000 2:0.000000
i:14 1:248.900606 2:0.000000
i:15 1:612.249343 2:0.000000
i:16 1:1000000.000000 2:0.000000
i:17 1:1000000.000000 2:0.000000
i:18 1:1000000.000000 2:0.000000
i:19 1:1000000.000000 2:0.000000
i:20 1:1000000.000000 2:0.000000
i:21 1:1000000.000000 2:0.000000
j and closest[j]:3 0
j and closest[j]:9 0
j and closest[j]:20 9
j and closest[j]:5 3
j and closest[j]:7 9
j and closest[j]:17 7
j and closest[j]:10 17
j and closest[j]:13 20
j and closest[j]:18 7
j and closest[j]:4 10
j and closest[j]:6 10

j and closest[j]:15 6
j and closest[j]:16 4
j and closest[j]:1 15
j and closest[j]:2 15
j and closest[j]:8 16
j and closest[j]:12 2
j and closest[j]:11 1
j and closest[j]:14 1
j and closest[j]:19 16
j and closest[j]:19 16
j and closest[j]:21 10
这里编码从 0 开始，总 cost 为 6723。
至于另一种情况下，在此不用演示了。

七、实验小结：

贪心算法本身是一种很重要的算法，但相对动态规划算法使用的场景较少。简单来说，适用于贪心算法的场景是：问题能够分解成子问题来解决，子问题的最优解能够递推到最终问题的最优解，这种子问题具有最优子结构。（也就是所谓的贪心选择性质和最优子结构）

贪心算法和动态规划的不同在于它对每一个子问题的解决方案都做出原则，不能回退。而动态规划作为“动态递归”，会递推+Memorization，保存以前的结果，并且根据以前的结果对当前进行选择，有回退的功能。所以我们动态规划往往是一层层来求解算出最后的答案，而贪心更像是“梭哈”，以一种直接的方式算出最后的答案。

本次实验简单跑了一下课堂上讲解的一些经典案例，但贪心算法本身的证明也很重要，在证明其正确性的时候我们也用到了很多推理证明的有关知识，这也是我在完成这次实验思考过了的问题。