第六章 上机

上机题

1, 假设非空二叉树采用二叉链存储结构, 所有结点值为单个字符且不相同。编写一个实验程序, 将一棵二叉树 bt 的左、右子树进行交换, 要求不破坏原二叉树, 并且采用相关数据进行测试。

2, 假设二叉树采用二叉链存储结构, 所有结点值为单个字符且不相同。编写一个实验程序, 求 x 和 y 结点的最近公共祖先结点（LCA）, 假设二叉树中存在结点值为和 y 的结点, 并且采用相关数据进行测试。

3. 假设一棵非空二叉树中的结点值为整数, 所有结点值均不相同。编写一个实验程序, 给出该二叉树的先序序列 pres 和中序序列 ins, 构造该二叉树的二叉链存储结构, 再给出其中两个不同的结点值 和 y, 输出这两个结点的所有公共祖先结点, 采用相关数据进行测试。

4. 假设二叉树采用二叉链存储结构, 所有结点值为单个字符且不相同。编写一个实验程序, 采用《教程》中例 6.16 的 3 种解法按层次顺序（从上到下、从左到右）输出一棵二叉树中的所有结点, 并且利用相关数据进行测试。

5. 假设二叉树采用二叉链存储结构, 所有结点值为单个字符且不相同。编写一个实验程序, 采用先序遍历和层次遍历方式输出二叉树中从根结点到每个叶予结点的路径, 并且利用相关数据进行测试。

6. 编写一个实验程序, 利用例 6.20 的数据（为了方便, 将该例中的所有权值扩大 100 倍）构造哈夫曼树和哈夫曼编码, 要求输出建立的哈夫曼树和相关哈夫曼编码。

解题思路

1. 解: 题目要求不破坏原二叉树, 只有通过交换二叉树 bt 的左、右子产生新的二叉树 bt1。假设 bt 的根结点为 b, btl 的根结点为 t, 对应的递归模型如下:

```
f(b.D)=t=None //若 b=None
f(b,t)=复制根结点 b 产生新结点t; //其他情况
    f(b.lchild,t1);
    f(b. rchild,t2);
    t.lchild=t2; t.rchild= t1 ;
```

对应的实验程序如下:

```
from BTree import BTree,BTNode
def Swap(bt):
    bt1=BTree()
    bt1.SetRoot(_Swap(bt.b))
    return bt1
def _Swap(b):
    if b==None:
        t=None
    else:
        t=BTNode(b.data)     # 复制根结点
        t1=_Swap(b.lchild)                    # 交换左子树
```

```
        t2=_Swap(b.rchild)                    # 交换右子树
        t.lchild=t2
        t.rchild=t1
    return t
# 主程序
b=BTNode('A')
p1=BTNode('B')
p2=BTNode('C')
p3=BTNode('D')
p4=BTNode('E')
p5=BTNode('F')
p6=BTNode('G')
b.lchild=p1
b.rchild=p2
p1.lchild=p3
p3.rchild=p6
p2.lchild=p4
p2.rchild=p5
bt=BTree()
bt.SetRoot(b)
print()
print("  bt: ",end=' '); print(bt.DispBTree())
print("  bt->bt1")
bt1=Swap(bt)
print("  bt1:",end=' '); print(bt1.DispBTree())
```

2. 解: 由于二叉树中存在结点值为 和 y 的结点，则 LCA 一定是存在的。采用先序历求根到 x 结点的路径 pathx（根到 x 结点的正向路径），根到 y 结点的路径 pathy，从头 F 始找到它们中最后一个相同的结点。类似地也可以采用后序非递归算法或者层次遍历。但这些方法要么需要遍历二叉树两次，要么回推路径比较麻烦。这里用先序遍历，一次遍历可完成。

设计 _LCA(t,x,y) 算法返回根结点为 t 的二叉树中 x 和 y 结点的 LCA:

(1) 若 t=None，返回 None。

(2) 若找到或者 y 结点，即 t.data==x or t.data==y，返回 t。

(3) 递归调用 p=_LCA(t.lchild,x,y), 在 t 的左子树中查找 x 或者 y 结点

(4) 递归调用 q=_LCA(t.rchild,x,y), 在 t 的右子树中查找 x 或者 y 结点

(5) 只有 p!=None 并且 q!=None(即在 t 的子树中找到 x 和 y 结点)，即 t 结点就是 LCA 时，才返回 t。

(6) 否则在找到 x 或者 y 结点的子树中继续查找另外一个结点

(7) 全部没有找到，返回 None。

对应的实验程序如下:

```
from BTree import BTree,BTNode
def LCA(bt,x,y):                    # 求 x 和 y 结点的 LCA
    return LCA1(bt.b,x,y).data
def LCA1(t,x,y):
    if t!=None:
        if t.data==x or t.data==y:       # 找到 x 或者 y 结点返回 t
            return t
        p=LCA1(t.lchild,x,y)             # 在左子树中查找 x 或者 y 结点
        q=LCA1(t.rchild,x,y)             # 在右子树中查找 x 或者 y 结点
        if p!=None and q!=None:          # 只有 t 的子树中找到 x 和 y 结点, 才返回 t
            return t
        if p!=None:
            return p
        if q!=None:
            return q
    return None
def solve(bt,x,y):
    print(" %c和%c的 LCA: %c" %(x,y,LCA(bt,x,y)))
# 主程序
b=BTNode('A')
p1=BTNode('B')
p2=BTNode('C')
p3=BTNode('D')
p4=BTNode('E')
p5=BTNode('F')
p6=BTNode('G')
b.lchild=p1
b.rchild=p2
p1.lchild=p3
p3.rchild=p6
p2.lchild=p4
p2.rchild=p5
bt=BTree()
bt.SetRoot(b)
print()
print(" bt: ",end=' '); print(bt.DispBTree())
solve(bt,'A','A')
solve(bt,'F','F')
solve(bt,'B','F')
solve(bt,'G','E')
solve(bt,'G','B')
solve(bt,'F','G')
```

3. 解: 由先序序列 pres 和中序序列 ins 构造二叉链的过程参见《教程》6.5.1 节。对于二叉链 bt 中两个不同的结点值 x 和 y, 用 ator 列表存放它们的所有公共祖先结点, 先求出它们的最近公共祖先结点 (求出后置 find 为 True), 当回退到 t 结点时若 find

为 True，说明结点 t 是公共祖先结点，将 t.data 添加到 ator 中。最后返回 ator。

```
class BTNode:                                          # 二叉链中结点类
    def __init__(self,d=None):                        # 构造方法
        self.data=d                                    # 结点值
        self.lchild=None                               # 左孩子指针
        self.rchild=None                               # 右孩子指针

class BTree:                                           # 二叉树类
    def __init__(self,d=None):                        # 构造方法
        self.b=None                                    # 根结点指针

    def DispBTree(self):                               # 返回二叉链的括号表示串
        return self._DispBTree(self.b)

    def _DispBTree(self,t):                            # 被 DispBTree 方法调用
        if t==None:                                    # 空树返回空串
            return ""
        else:
            bstr=str(t.data)                                   # 输出根结点值
            if t.lchild!=None or t.rchild!=None:
                bstr+="("                              # 有孩子结点时输出"("
                bstr+=self._DispBTree(t.lchild)        # 递归输出左子树
                if t.rchild!=None:
                    bstr+=","                          # 有右孩子结点时输出","
                bstr+=self._DispBTree(t.rchild)        # 递归输出右子树
                bstr+=")"                              # 输出")"
            return bstr

def CreateBTree1(pres,ins):                            # 由先序序列 pres 和中序序列 ins 构造二叉
    bt=BTree()
    bt.b=_CreateBTree1(pres,0,ins,0,len(pres))
    return bt

def _CreateBTree1(pres,i,ins,j,n):                     # 被 CreateBTree1 调用
    if n<=0: return None
    d=pres[i]                                          # 取根结点值 d
    t=BTNode(d)                                        # 创建根结点 (结点值为 d)
    p=ins.index(d)                                     # 在 ins 中找到根结点的索引
    k=p-j                                              # 确定左子树中结点个数 k
    t.lchild=_CreateBTree1(pres,i+1,ins,j,k)           # 递归构造左子树
    t.rchild=_CreateBTree1(pres,i+k+1,ins,p+1,n-k-1)   # 递归构造右子树
    return t

def CA(bt,x,y):                                        # 在 bt 中求 x 和 y 的所有公共祖先结点
    global find
```

```
    find=False                                          # 表示是否找到 x 和 y 的最近公共祖先
    ator=[]                                            # 存放 x 和 y 的所有公共祖先结点
    _CA(bt.b,x,y,ator)
    return ator

def _CA(t,x,y,ator):                                   # 被 CA() 函数调用
    global find
    if t==None: return None
    if t.data==x or t.data==y:
        return t
    left=_CA(t.lchild,x,y,ator)
    right=_CA(t.rchild,x,y,ator)
    if left and right:
        find=True
        ator.append(t.data)
        return t
    if left!=None:
        if find: ator.append(t.data)
        return left
    if right!=None:
        if find: ator.append(t.data)
        return right
    return None

# 主程序
pres=[2,1,3,4,5,8,9,13,10,12,7,11,6]
ins=[3,1,5,4,2,10,13,9,7,12,8,11,6]
bt=CreateBTree1(pres,ins)
print()
print("  bt;",end=' ')
print(bt.DispBTree())
x,y=5,10
print("  (1)%2d和%2d的所有公共祖先:" %(x,y),end=' ')
print(CA(bt,x,y))
x,y=6,7
print("  (2)%2d和%2d的所有公共祖先:" %(x,y),end=' ')
print(CA(bt,x,y))
x,y=3,4
print("  (3)%2d和%2d的所有公共祖先:" %(x,y),end=' ')
print(CA(bt,x,y))
x,y=10,7
print("  (4)%2d和%2d的所有公共祖先:" %(x,y),end=' ')
print(CA(bt,x,y))
```

4. 解: 采用层次遍历, 难点是如何确定每一层结点访问完, 3 种解法见《教程》中例 6.16. 对应的实验程序如下:

```
from BTree import BTree,BTNode
from collections import deque

class QNode:                                    # 队列元素类
    def __init__(self,l,p):                     # 构造方法
        self.lev=l                              # 结点的层次
        self.node=p                             # 结点引用
def Leveldisp1(bt):                             # 解法 1: 按分层次顺序输出所有结点
    qu=deque()                                  # 定义一个队列 qu
    curl=1                                      # 当前层次, 从 1 开始
    strl=""
    qu.append(QNode(1,bt.b))                    # 根结点 (层次为 1) 进队
    while len(qu)>0:                            # 队不空循环
        p=qu.popleft()                          # 出队一个结点
        if p.lev==curl:                         # 当前结点的层次为 curl 大于 k, 返回
            strl+=p.node.data+" "
        else:
            print("     第"+str(curl)+" 层结点: "+strl)
            curl+=1
            strl=""
            strl+=p.node.data+" "               # 当前结点是第 curl+1 层的首结点
        if p.node.lchild!=None:                 # 有左孩子时将其进队
            qu.append(QNode(p.lev+1,p.node.lchild))
        if p.node.rchild!=None:                 # 有右孩子时将其进队
            qu.append(QNode(p.lev+1,p.node.rchild))
    print("     第"+str(curl)+" 层结点: "+strl)
def Leveldisp2(bt):                             # 解法 2: 按分层次顺序输出所有结点
    qu=deque()                                  # 定义一个队列 qu
    curl=1                                      # 当前层次, 从 1 开始
    strl=""
    last=bt.b                                   # 第 1 层最右结点
    qu.append(bt.b)                             # 根结点进队
    while len(qu)>0:                            # 队不空循环
        p=qu.popleft()                          # 出队一个结点
        strl+=p.data+" "                        # 当前结点是第 curl 层的结点
        if p.lchild!=None:                      # 有左孩子时将其进队
            q=p.lchild
            qu.append(q)
        if p.rchild!=None:                      # 有右孩子时将其进队
            q=p.rchild
            qu.append(q)
        if p==last:                             # 当前层的所有结点处理完毕
            print("     第"+str(curl)+" 层结点: "+strl);
            strl=""
            last=q                              # 让 last 指向下一层的最右结点
            curl+=1
```

```
def Leveldisp3(bt):                                 # 解法 3: 按分层次顺序输出所有结点
    qu=deque()                                      # 定义一个队列 qu
    curl=1                                          # 当前层次, 从 1 开始
    qu.append(bt.b)                                 # 根结点进队
    strl=bt.b.data
    while len(qu)>0:                                # 队不空循环
        print("     第"+str(curl)+" 层结点: "+strl)
        strl=""
        n=len(qu)                                   # 求出当前层结点个数
        for i in range(n):                          # 出队当前层的 n 个结点
            p=qu.popleft()                          # 出队一个结点
            if p.lchild!=None:                      # 有左孩子时将其进队
                qu.append(p.lchild)
                strl+=p.lchild.data+" "
            if p.rchild!=None:                      # 有右孩子时将其进队
                qu.append(p.rchild)
                strl+=p.rchild.data+" "
        curl+=1                                     # 转向下一层
    return 0

def solve(bt):
    print()
    print(" 求解结果");
    print("   解法 1:");
    Leveldisp1(bt);
    print("   解法 2:");
    Leveldisp2(bt);
    print("   解法 3:");
    Leveldisp3(bt);

# 主程序
b=BTNode('A')
p1=BTNode('B')
p2=BTNode('C')
p3=BTNode('D')
p4=BTNode('E')
p5=BTNode('F')
p6=BTNode('G')
b.lchild=p1
b.rchild=p2
p1.lchild=p3
p3.rchild=p6
p2.lchild=p4
p2.rchild=p5
bt=BTree()
bt.SetRoot(b)
```

```
print("bt:",end='  ');print(bt.DispBTree())
solve(bt)
```

5. 解: 先序遍历求解思路见《教程》中例 6.15 的解法 2, 层次遍历求解思路见《教程》中的例 6.17。对应的实验程序如下:

```
from BTree import BTree,BTNode
from collections import deque
def AllPath1(bt):                                   # 解法 1: 先序遍历
    path=[None]*100
    d=-1
    PreOrder(bt.b,path,d)

def PreOrder(t,path,d):                             # 先序遍历输出结果
    if t!=None:
        if t.lchild==None and t.rchild==None:    #t 为叶子结点
            print("     根结点到%c的路径: " %(t.data),end=' ')
            for i in range(d+1):
                print(path[i]+" ",end=' ')
            print(t.data)
        else:
            d+=1; path[d]=t.data                    # 将当前结点放入路径中
            PreOrder(t.lchild,path,d)               # 递归遍历左子树
            PreOrder(t.rchild,path,d)               # 递归遍历右子树

class QNode:                                        # 队列元素类
    def __init__(self,p,pre):                       # 构造方法
        self.node=p                                 # 当前结点引用
        self.pre=pre                                # 当前结点的双亲结点

def AllPath2(bt):                                   # 解法 2: 层次遍历
    qu=deque()                                      # 定义一个队列 qu
    qu.append(QNode(bt.b,None))                     # 根结点 (双亲为 None) 进队
    while len(qu)>0:                                # 队不空循环
        p=qu.popleft()                              # 出队一个结点
        if p.node.lchild==None and p.node.rchild==None: #p 为叶子结点
            res=[]
            res.append(p.node.data)
            q=p.pre                                 #q 为双亲
            while q!=None:                          # 找到根结点为止
                res.append(q.node.data)
                q=q.pre
            print("     根结点到%c的路径: " %(p.node.data),end=' ')
            res.reverse()                           # 逆置 res
            print('  '.join(res))
        if p.node.lchild!=None:                     # 有左孩子时将其进队
            qu.append(QNode(p.node.lchild,p))    # 置其双亲为 p
```

```
        if p.node.rchild!=None:                  # 有右孩子时将其进队
            qu.append(QNode(p.node.rchild,p))    # 置其双亲为 p
# 主程序
b=BTNode('A')
p1=BTNode('B')
p2=BTNode('C')
p3=BTNode('D')
p4=BTNode('E')
p5=BTNode('F')
p6=BTNode('G')
b.lchild=p1
b.rchild=p2
p1.lchild=p3
p3.rchild=p6
p2.lchild=p4
p2.rchild=p5
bt=BTree()
bt.SetRoot(b)
print()
print("  bt:",end='  ');print(bt.DispBTree())
print("  解法 1")
AllPath1(bt)
print("  解法 2")
AllPath2(bt)
```

6. 解: 构造哈夫曼树和哈夫曼编码的原理参见《教程》6.7 节。对应的实验程序如下:

```
import heapq                                     # 导入优先队列模块
class HTNode:                                    # 哈夫曼树结点类
    def __init__(self,d=" ",w=None):            # 构造方法
        self.data=d                              # 结点值
        self.weight=w                            # 权值
        self.parent=-1                           # 指向双亲结点
        self.lchild=-1                           # 指向左孩子结点
        self.rchild=-1                           # 指向右孩子结点
        self.flag=True                           # 标识是双亲的左 (True) 或者右 (False) 孩子

def CreateHT():                                  # 构造哈夫曼树
    global ht                                    # 全局列表, 存放哈夫曼树
    global n0
    global D
    global W
    ht=[None]*(2*n0-1)                           # 初始为含 2n0-1 个空结点
    heap=[]                                      # 优先队列元素为 [w,i], 按 w 权值建立小根堆
    for i in range(n0):                          #i 从 0 到 n0-1 循环建立 n0 个叶子结点并进队
        ht[i]=HTNode(D[i],W[i])                  # 建立一个叶子结点
        heapq.heappush(heap,[W[i],i])            # 将 [W[i],i] 进队
```

```
    for i in range(n0,2*n0-1):                  #i 从 n0 到 2n0-2 循环做 n0-1 次合并操作
        p1=heapq.heappop(heap)                  # 出队两个权值最小的结点 p1 和 p2
        p2=heapq.heappop(heap)
        ht[i]=HTNode()                          # 新建 ht[i] 结点
        ht[i].weight=ht[p1[1]].weight+ht[p2[1]].weight  # 求权值和
        ht[p1[1]].parent=i                      # 设置 p1 的双亲为 ht[i]
        ht[i].lchild=p1[1]                      # 将 p1 作为双亲 ht[i] 的左孩子
        ht[p1[1]].flag=True
        ht[p2[1]].parent=i                      # 设置 p2 的双亲为 ht[i]
        ht[i].rchild=p2[1]                      # 将 p2 作为双亲 ht[i] 的右孩子
        ht[p2[1]].flag=False
        heapq.heappush(heap,[ht[i].weight,i])   # 将新结点 ht[i] 进队

def DispHT():                                    # 输出哈夫曼树
    global n0
    global ht
    print("    i      ",end=' ')
    for i in range(2*n0-1):
        print("%3d" %(i),end=' ')
    print()
    print("    D[i]   ",end=' ')
    for i in range(2*n0-1):
        print("%3s" %(ht[i].data),end=' ')
    print()
    print("    W[i]   ",end=' ')
    for i in range(2*n0-1):
        print("%3g" %(ht[i].weight),end=' ')
    print()
    print("    parent ",end=' ')
    for i in range(2*n0-1):
        print("%3d" %(ht[i].parent),end=' ')
    print()
    print("    lchild ",end=' ')
    for i in range(2*n0-1):
        print("%3d" %(ht[i].lchild),end=' ')
    print()
    print("    rchild ",end=' ')
    for i in range(2*n0-1):
        print("%3d" %(ht[i].rchild),end=' ')
    print()

def CreateHCode():                              # 根据哈夫曼树求哈夫曼编码
    global n0
    global ht
    global hcd                                   # 全局列表，存放哈夫曼编码
    hcd=[]
```

```
    for i in range(n0):                    # 遍历下标从 0 到 n0-1 的叶子结点
        code=[]
        j=i                                # 从 ht[i] 开始找双亲结点
        while ht[j].parent!=-1:
            if ht[j].flag:                 #ht[j] 结点是双亲的左孩子
                code.append("0")
            else:                          #ht[j] 结点是双亲的右孩子
                code.append("1")
            j=ht[j].parent
        code.reverse()
        hcd.append(''.join(code))          # 将 code 转换为字符串并添加到 hcd 中
def DispHCode():                           # 输出哈夫曼编码
    global hcd
    for i in range(len(hcd)):
        print("    "+ht[i].data+": "+hcd[i])
if __name__ == '__main__':
    n0=8                                               # 编码的字符个数
    D=['a','b','c','d','e','f','g','h']                # 字符列表
    W=[7,19,2,6,32,3,21,10]                            # 权值列表
    print()
    print("  (1) 建立哈夫曼树")
    CreateHT()
    print("  (2) 输出哈夫曼树")
    DispHT()
    print("  (3) 建立哈夫曼编码")
    CreateHCode()
    print("  (4) 输出哈夫曼编码")
    DispHCode()
```

第七章 上机

上机题

1. 有一个文本文件 gin.txt 存放一个带权无向图的数据, 第一行为 n 和 e, 分别为顶点个数和边数, 接下来的 e 行每行为 u,v,w, 表示顶点 “到 的边的权值为 w, 如以下数据表示如图 2.52 所示的图 (任意两个整数之间用空格分隔):

```
6 8
0 1 2
0 2 2
0 3 5
1 3 1
2 3 6
3 4 5
3 5 2
4 5 1
```

编写一个实验程序, 利用文件 gin.txt 中的图求出顶点 0 到顶点 4 的所有路径及其路径长度。

2. 编写一个实验程序, 利用文件 gin.txt 中的图求出顶点 0 到顶点 5 的经过边数最少的一条路径及其路径长度。

3. 编写一个实验程序, 利用文件 gin.xt 中的图采用 Prim 算法求出以顶点 0 为起始顶点的一棵最小生成树。

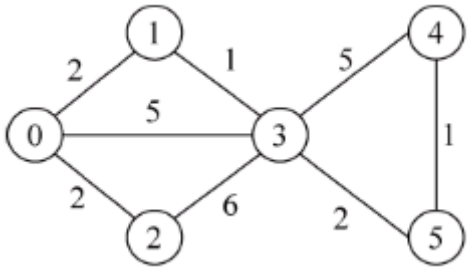

Figure 1: 一个带权无向图

4. 编写一个实验程序, 利用文件 gin.txt 中的图采用 Kruskal 算法求出一棵最小生成树。

5. 编写一个实验程序, 利用文件 gin.txt 中的图求出以顶点 0 为源点的所有单源最短路径及其长度。

6. 编写一个实验程序, 利用文件 gin,txt 中的图求出所有两个顶点之间的最短路径及其长度。

7. 有一片大小为 m × n (m,n $\leq$ 100) 的森林, 其中有若干群猴子, 数字 0 表示树, 1 表示猴子, 凡是由 0 或者矩形围起来的区域表示有一个猴群在这一带。编写一个实验程序, 求一共有多少个猴群及每个猴群的数量。森林用二维数组 g 表示, 要求按递增顺序输出猴群的数量, 并用相关数据进行测试。

8. 最优配餐问题。栋栋最近开了一家餐饮连锁店, 提供外卖服务, 随着连锁店越来越多, 怎么合理地给客户送餐成为一个急需解决的问题。

栋栋的连锁店所在的区域可以看成是一个 n×n 的方格图 (如图所示), 方格的格点上的位置可能包含栋栋的分店 (绿色标注) 或者客户 (蓝色标注), 有一些格点是不能经过的 (红色标注)。

方格图中的线表示可以行走的道路, 相邻两个格点的距离为 1。栋栋要送餐必须走可以行走的道路, 而且不能经过红色标注的点。

送餐的主要成本体现在路上所花的时间, 每份餐每走一个单位的距离需要花费一元钱每个客户的需求都可以由栋栋的任意分店配送, 每个分店没有配送总量的限制。

现在有栋栋的客户的需求, 请问在最优的送餐方式下送这些餐需要花费多大的成本?

输入格式: 输入的第一行包含 4 个整数 m, n, k, d, 分别表示方格图的大小、栋栋的分店数量、客户的数量, 以及不能经过的点的数量; 接下来 m 行, 每行两个整数 x_i, y_i, 表示栋栋的一个分店在方格图中的横坐标和纵坐标; 接下来 k 行, 每行 3 个整数 x_i, y_i, c_i, 分别表

示每个客户在方格图中的横坐标、纵坐标和订餐的量（注意，可能有多个客户在方格图中的同一个位置）；接下来 d 行，每行两个整数，分别表示每个不能经过的点的横坐标和纵坐标。

输出格式：输出一个整数，表示最优送餐方式下所需要花费的成本。

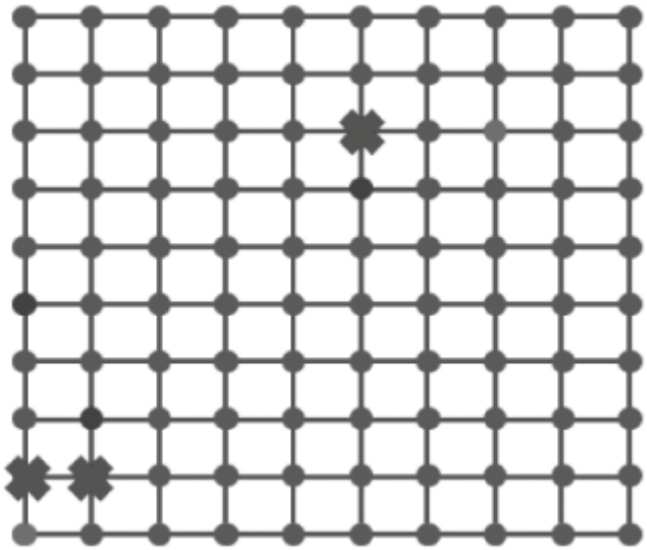

Figure 2: 一个方格图

样例输入:

```
10 2 3 3
1 1 //第 1 个分店位置
8 8 //第 2 个分店位置
1 5 1 //第 1 个客户位置和订餐量
2 3 3 //第 2 个客户位置和订餐量
6 7 2 //第 3 个客户位置和订餐量
1 2 //第 1 个不能走的位置
2 2 //第 2 个不能走的位置
6 8 //第 3 个不能走的位置
```

样例输出:

```
29
```

解题思路

1. 解: 这里采用邻接矩阵存储图, 设计 MatGraph 类用于读取 gin.txt 文件并创建图的存储结构。对应的 ExpMatGraph.py 程序文件如下:

```
MAXV=100                                    # 表示最多顶点个数
INF=0x3f3f3f3f                              # 表示 ∞
```

```
visited=[0]*MAXV                            # 全局访问标志数组
class MatGraph:                             # 图邻接矩阵类
    def __init__(self,n=0,e=0):             # 构造方法
        self.n=n                            # 顶点数
        self.e=e                            # 边数
        self.edges=[[INF]*MAXV for i in range(MAXV)]    # 邻接矩阵数组
        for i in range(MAXV):               # 主对角线元素置为 0
            self.edges[i][i]=0
    def CreateMatGraph(self):               # 通过文件数据建立图的邻接矩阵
        f=open("gin.txt","r")
        tmp=f.readline().split()            # 读取第 1 行
        self.n=int(tmp[0])
        self.e=int(tmp[1])
        while True:
            tmp=f.readline().split()
            if not tmp: break
            i,j,w=int(tmp[0]),int(tmp[1]),int(tmp[2])
            self.edges[i][j]=w
            self.edges[j][i]=w
        f.close()
    def DispMatGraph(self):                 # 输出图
        for i in range(self.n):
            for j in range(self.n):
                if self.edges[i][j]==INF:
                    print("%4s"%("∞"),end=' ')
                else:
                    print("%5d" %(self.edges[i][j]),end=' ')
            print()
```

采用带回溯的深度优先遍历方法求解, 设计思路参见《教程》第 7 章例 7.7。对应的实验程序如下:

```
from ExpMatGraph import MatGraph,INF,MAXV
cnt=0                                       # 路径条数
visited=[0]*MAXV                            # 全局访问标志数组

def FindallPath(g,u,v):                     # 求 u 到 v 的所有简单路径
    path=[-1]*MAXV
    d=-1                                    #path[0..d] 存放一条路径
    sum=0                                   # 存放路径长度
    for i in range(g.n): visited[i]=0       # 初始化
    FindallPath1(g,u,v,path,d,sum)

def FindallPath1(g,u,v,path,d,sum):         # 被 Findallpath 调用
    global cnt
    visited[u]=1
    d+=1; path[d]=u                         # 顶点 u 加入到路径中
```

```
    if u==v:                                        # 找到一条路径后输出
        cnt+=1
        print("    第%d条路径:" %(cnt),end=' ')
        for i in range(d+1):
            print(path[i],end='  ')
        print("\t长度:",sum)                    # 输出一条路径

    for w in range(g.n):                            # 处理顶点 u 的所有出边
        if g.edges[u][w]!=0 and g.edges[u][w]!=INF:
            if visited[w]==0:                       #w 没有访问过
                FindallPath1(g,w,v,path,d,sum+g.edges[u][w])  # 递归调用
    visited[u]=0                                    # 回溯, 重置 visited[u] 为 0

# 主程序
g=MatGraph()
g.CreateMatGraph()
print()
print("  图 g:")
g.DispMatGraph()
u,v=0,4
print("  %d到%d的所有路径:" %(u,v))
FindallPath(g,u,v)
```

2. 解: 采用广度优先遍历方法求解, 设计思路参见《教程》第 7 章例 7.9。对应的实验程序如下:

```
from ExpMatGraph import MatGraph,INF,MAXV
from collections import deque
visited=[0]*MAXV                                    # 全局访问标志数组

class QNode:                                        # 队列元素类
    def __init__(self,p,pre):                       # 构造方法
        self.vno=p                                  # 当前顶点编号
        self.pre=pre                                # 当前结点的前驱结点

def ShortPath(G,u,v):                               # 求 u 到 v 的一条最短简单路径
    res=[]                                          # 存放结果
    qu=deque()                                      # 定义一个队列 qu
    qu.append(QNode(u,None))                        # 起始点 u(前驱为 None) 进队
    visited[u]=1                                    # 置已访问标记
    while len(qu)>0:                                # 队不空循环
        p=qu.popleft()                              # 出队一个结点
        if p.vno==v:                                # 当前结点 p 为 v 结点
            res.append(v)
            q=p.pre                                 #q 为前驱结点
            while q!=None:                          # 找到起始结点为止
                res.append(q.vno)
```

```
            q=q.pre
        res.reverse()                           # 逆置 res 构成正向路径
        return res
    for w in range(g.n):
        if g.edges[p.vno][w]!=0 and g.edges[p.vno][w]!=INF:
            if visited[w]==0:                   # 存在边 <v,w> 并且 w 未访问
                qu.append(QNode(w,p))           # 置其前驱结点为 p
                visited[w]=1                    # 置已访问标记

# 主程序
g=MatGraph()
g.CreateMatGraph()
print()
print("  图 g:")
g.DispMatGraph()
u,v=0,5
print("  %d到%d的经过边最少的路径:" %(u,v),end=' ')
print(ShortPath(g,u,v))
```

3. 解: 采用 Prim 算法直接求解, 设计思路参见《教程》7.5.2 节。对应的实验程序如下:

```
from ExpMatGraph import MatGraph,INF,MAXV
def Prim(g,v):                                  # 求最小生成树
    lowcost=[0]*MAXV                            # 建立数组 lowcost
    closest=[0]*MAXV                            # 建立数组 closest
    sum=0                                       # 存放权值和
    for i in range(g.n):                        # 给 lowcost[] 和 closest[] 置初值
        lowcost[i]=g.edges[v][i]
        closest[i]=v
    for i in range(1,g.n):                      # 找出最小生成树的 n-1 条边
        min=INF
        k=-1
        for j in range(g.n):                    # 在 (V-U) 中找出离 U 最近的顶点 k
            if lowcost[j]!=0 and lowcost[j]<min:
                min=lowcost[j]
                k=j                             #k 记录最小顶点的编号
        print("      (%d,%d): %d" %(closest[k],k,+min))     # 输出最小生成树的边
        sum+=min                                # 累计权值和
        lowcost[k]=0                            # 将顶点 k 加入 U 中
        for j in range(g.n):                    # 修改数组 lowcost 和 closest
            if lowcost[j]!=0 and g.edges[k][j]<lowcost[j]:
                lowcost[j]=g.edges[k][j]
                closest[j]=k
    print("   所有边的取值和 =%d" %(sum))


# 主程序
```

```
g=MatGraph()
g.CreateMatGraph()
print()
print("  图 g:")
g.DispMatGraph()
v=0
print("  求出的一棵最小生成树");
Prim(g,0)
```

4. 解: 采用基本的 Kruskal 算法求解, 设计思路参见《教程》7.5.3 节。对应的实验程序如下:

```
from ExpMatGraph import MatGraph,INF,MAXV
from operator import itemgetter,attrgetter
# 基本的 Kruskal 算法
def Kruskal(g):                                  # 求最小生成树
    vset=[-1]*MAXV                               # 建立数组 vset
    sum=0                                        # 存放权值和
    E=[]                                         # 建立存放所有边的列表 E
    for i in range(g.n):                         # 由邻接矩阵 g 产生的边集数组 E
        for j in range(i+1,g.n):                 # 对于无向图仅考虑上三角部分的边
            if g.edges[i][j]!=0 and g.edges[i][j]!=INF:
                E.append([i,j,g.edges[i][j]])    # 添加 [i,j,w] 元素
    E.sort(key=itemgetter(2))                    # 按权值递增排序
    for i in range(g.n):vset[i]=i                # 初始化辅助数组
    cnt=1                                        #cnt 表示当前构造生成树的第几条边, 初值为 1
    j=0                                          # 取 E 中边的下标, 初值为 0
    while cnt<g.n:                               # 生成的边数小于 n 时循环
        u1,v1=E[j][0],E[j][1]                    # 取一条边的头尾顶点
        sn1=vset[u1]
        sn2=vset[v1]                             # 分别得到两个顶点所属的集合编号
        if sn1!=sn2:                             # 两顶点属于不同的集合, 加入不会构成回路
            print("   (%d,%d):%d" %(u1,v1,E[j][2]))     # 输出最小生成树的边
            sum+=E[j][2]                         # 累计权值和

            cnt+=1                               # 生成边数增 1
            for i in range(g.n):                 # 两个集合统一编号
                if vset[i]==sn2:                 # 集合编号为 sn2 的改为 sn1
                    vset[i]=sn1
        j+=1                                     # 继续取 E 的下一条边
    print("   所有边的取值和 =%d" %(sum))


# 主程序
g=MatGraph()
g.CreateMatGraph()
print()
```

```
print("  图 g:")
g.DispMatGraph()
v=0
print("  求出的一棵最小生成树");
Kruskal(g)
```

5. 解: 采用 Diikstra 算法求解，设计思路参见《教程》7.6.2 节。对应的实验程序如下:

```
from ExpMatGraph import MatGraph,INF,MAXV
def Dijkstra(g,v):                          # 求从 v 到其他顶点的最短路径
    dist=[-1]*MAXV                          # 建立 dist 数组
    path=[-1]*MAXV                          # 建立 path 数组
    S=[0]*MAXV                              # 建立 S 数组
    for i in range(g.n):
        dist[i]=g.edges[v][i]               # 最短路径长度初始化
        if g.edges[v][i]<INF:               # 最短路径初始化
            path[i]=v                       #v 到 i 有边，置路径上顶点 i 的前驱为 v
        else:                               #v 到 i 没边时，置路径上顶点 i 的前驱为-1
            path[i]=-1
    S[v]=1                                  # 源点 v 放入 S 中
    u=-1
    for i in range(g.n-1):                  # 循环向 S 中添加 n-1 个顶点
        mindis=INF                          #mindis 置最小长度初值
        for j in range(g.n):                # 选取不在 S 中且具有最小距离的顶点 u
            if S[j]==0 and dist[j]<mindis:
                u=j
                mindis=dist[j]
        S[u]=1                              # 顶点 u 加入 S 中
        for j in range(g.n):                # 修改不在 s 中的顶点的距离
            if S[j]==0:                     # 仅仅修改 S 中的顶点 j
                if g.edges[u][j]<INF and dist[u]+g.edges[u][j]<dist[j]:
                    dist[j]=dist[u]+g.edges[u][j]
                    path[j]=u
    DispAllPath(dist,path,S,v,g.n)          # 输出所有最短路径及长度

def DispAllPath(dist,path,S,v,n):           # 输出从顶点 v 出发的所有最短路径
    for i in range(n):                      # 循环输出从顶点 v 到 i 的路径
        if S[i]==1 and i!=v:
            apath=[]
            print("    从%d到%d最短路径长度: %d \t路径:" %(v,i,dist[i]),end=' ')
            apath.append(i)                 # 添加路径上的终点
            k=path[i];
            if k==-1:                       # 没有路径的情况
                print(" 无路径")
            else:                           # 存在路径时输出该路径
                while k!=v:
                    apath.append(k)         # 顶点 k 加入到路径中
```

```
            k=path[k]
        apath.append(v)                 # 添加路径上的起点
        apath.reverse()                 # 逆置 apath
        print(apath)                    # 输出最短路径


# 主程序
g=MatGraph()
g.CreateMatGraph()
print()
print("  图 g:")
g.DispMatGraph()
v=0
print("  求解结果")
Dijkstra(g,v)
```

6. 解: 采用 Floyd 算法求解, 设计思路参见《教程》7.6.3 节。对应的实验程序如下:

```
from ExpMatGraph import MatGraph,INF,MAXV
def Floyd(g):                           # 输出所有两个顶点之间的最短路径
    A=[[0]*MAXV for i in range(MAXV)]   # 建立 A 数组
    path=[[0]*MAXV for i in range(MAXV)] # 建立 path 数组
    for i in range(g.n):                # 给数组 A 和 path 置初值即求 A-1[i][j]
        for j in range(g.n):
            A[i][j]=g.edges[i][j]
            if i!=j and g.edges[i][j]<INF:
                path[i][j]=i            #i 和 j 顶点之间有边时
            else:
                path[i][j]=-1           #i 和 j 顶点之间没有边时
    for k in range(g.n):                # 求 Ak[i][j]
        for i in range(g.n):
            for j in range(g.n):
                if A[i][j]>A[i][k]+A[k][j]:
                    A[i][j]=A[i][k]+A[k][j]
                    path[i][j]=path[k][j]   # 修改最短路径
    Dispath(A,path,g)                   # 生成最短路径和长度

def Dispath(A,path,g):                  # 输出所有的最短路径和长度
    for i in range(g.n):
        for j in range(g.n):
            if A[i][j]!=INF and i!=j:   # 若顶点 i 和 j 之间存在路径
                print("    顶点%d到%d的最短路径长度: %d\t路径:" %(i,j,A[i][j]),end='')
                k=path[i][j]
                apath=[j]               # 路径上添加终点
                while k!=-1 and k!=i:   # 路径上添加中间点
                    apath.append(k)     # 顶点 k 加入到路径中
                    k=path[i][k]
```

```
                apath.append(i)                 # 路径上添加起点
                apath.reverse()                 # 逆置
                print(apath)                    # 输出最短路径


# 主程序
g=MatGraph()
g.CreateMatGraph()
print()
v=0
print("  求解结果")
Floyd(g)
```

7. 解: 从 g[i][j]=1 的 (i,j) 位置出发遍历上、下、左、右 4 个方位为 1 的个数 (面积), 可以采用深度优先和广度优先遍历求解。对应的实验程序如下:

```
from collections import deque               # 引用双端队列 deque
MAXV=100
dx=[1,0,-1,0]                               #x 方向偏移量
dy=[0,1,0,-1]                               #y 方向偏移量
# 解法 1: 采用深度优先遍历求解
def solve1(g):
    m=len(g)
    n=len(g[0])
    ans=[]
    for i in range(m):
        for j in range(n):
            if g[i][j]==1:
                area=dfs(g,m,n,i,j)         # 求 (i,j) 出发遍历的面积
                if area!=0:
                    ans.append(area)        # 将面积添加到 ans 中
    return ans

def dfs(g,m,n,i,j):                         #(i,j) 位置出发深度优先遍历
    g[i][j]=-1
    area=1                                  #(i,j) 位置计入面积
    for k in range(4):                      # 考虑上下左右 4 个方位
        x=i+dx[k]
        y=j+dy[k]
        if x>=0 and x<m and y>=0 and y<n and g[x][y]==1:
            area+=dfs(g,m,n,x,y)            # 累计面积
    return area

# 解法 2: 采用广度优先遍历求解
def solve2(g):
    m=len(g)
    n=len(g[0])
```

```
    ans=[]
    for i in range(m):
        for j in range(n):
            if g[i][j]==1:
                area=bfs(g,m,n,i,j)          # 求 (i,j) 出发遍历的面积
                if area!=0:
                    ans.append(area)         # 将面积添加到 ans 中
    return ans

def bfs(g,m,n,i,j):                          #(i,j) 位置出发广度优先遍历
    qu=deque()                               # 将双端队列作为普通队列 qu
    g[i][j]=-1
    area=1                                   #(i,j) 位置计入面积
    qu.append([i,j])                         #(i,j) 进队
    while len(qu)>0:                         # 队不空循环
        v=qu.popleft()                       # 出队顶点 v
        for k in range(4):                   # 考虑上下左右 4 个方位
            x=v[0]+dx[k]
            y=v[1]+dy[k]
            if x>=0 and x<m and y>=0 and y<n and g[x][y]==1:
                area+=1                      # 累计面积
                g[x][y]=-1                   # 置已访问标记
                qu.append([x,y])
    return area

# 主程序
g=[[0,1,1,1,1,0,0,0,1,1],
   [1,0,1,1,1,1,0,1,0,0],
   [1,0,1,1,1,0,0,1,1,1],
   [0,0,0,0,0,0,0,0,1,1]]
print()
print("  解法 1(DFS)")
visited=[0]*MAXV
ans=solve1(g)
ans.sort()
print("     共有%d个猴群" %(len(ans)))
print("     各猴群的数量:",ans)
for i in range(len(g)):                      # 恢复 g
    for j in range(len(g[0])):
        if g[i][j]==-1: g[i][j]=1
print("  解法 2(BFS)")
visited=[0]*MAXV
ans=solve2(g)
ans.sort()
print("     共有%d个猴群" %(len(ans)))
print("     各猴群的数量:",ans)
```

8. 解: 采用广度优先遍历从分店搜索客户 (从一个分店出发可以给多个客户送餐), 用 ans 存放所需要花费的成本 (初始为 0)。先将所有分店进队 (每个分店看成一个搜索点), 再出队一个搜索点, 找到所有相邻可走的搜索点并进队。若搜索点是客户, 计算花费的成本并累加到 ans 中。简单地说, 采用多个初始搜索点 (分店) 同步搜索的广度优先遍历, 由于是同步。最先找到的客户的送餐成本一定是最小的。一旦找到一个新搜索点 (含客户), 将其看成是一个分店继续搜索。处理不能经过的点十分简单, 仅将对应位置的访问标记数组 vis 的元素值置为 1 即可。对应的实验程序如下:

```
from collections import deque              # 引用双端队列 deque
dx=[1,0,-1,0]                              #x 方向偏移量
dy=[0,1,0,-1]                              #y 方向偏移量
qu=deque()                                 # 将双端队列作为普通队列 qu

class QNode:                               # 搜索点（队元素）类型
    def __init__(self,x1,y1,d1=0):         # 构造方法
        self.x=x1
        self.y=y1
        self.dep=d1

def Init():                                # 初始化
    global n,k
    global A,vis,cost
    n,m,k,d=map(int,input().split())
    N=n+2
    A=[[0]*N for i in range(N)]          # 方格图
    vis=[[0]*N for i in range(N)]        # 访问标记
    cost=[[0]*N for i in range(N)]       #(x,y) 位置的订餐量\
    for i in range(1,m+1):                 # 输入分店
        x,y=map(int,input().split())
        e=QNode(x,y)
        qu.append(e)                       # 分店进队
        vis[x][y]=1
    for i in range(1,k+1):                 # 输入客户位置和订餐量
        x,y,c=map(int,input().split())
        A[x][y]=1                          # 客户位置设置为 1
        cost[x][y]+=c                      # 累计相同客户位置的订餐量
    for i in range(1,d+1):                 # 输入不能走的位置
        x,y=map(int,input().split())
        vis[x][y]=1                        # 不能走的位置设置 vis 为 1

def BFS():                                 # 求解算法
    global n,k,cnt,ans
    global A,vis,cost
    while qu:                              # 队不空循环
        if cnt==k: return
        e=qu.popleft()                     # 出队元素 e
```

```
        sx,sy,d=e.x,e.y,e.dep
        for i in range(4):                  # 找搜索点的相邻可走搜索点 (ex,ey)
            ex=sx+dx[i]
            ey=sy+dy[i]
            if vis[ex][ey]==0 and ex>=1 and ex<=n and ey>=1 and ey<=n:
                vis[ex][ey]=1
                e1=QNode(ex,ey,d+1)
                qu.append(e1)               # 搜索点 (ex,ey,d+1) 进队
                if A[ex][ey]==1:            # 找到一个客户
                    ans+=e1.dep*cost[ex][ey]
                    cnt+=1
# 主程序
Init()
ans=0
cnt=0
BFS()
print(ans)
```