# 贪心算法求解d树

1. 问题描述

给定一棵带权树T和路径长度d，T的每个边的权都为整数。树T的一个顶点子集S，从T中删除S中的所有节点，则得到一个森林T/S。如果T/S中所有书从根到叶子节点的路径长度都不超过d，则称T/S是一个d森林。本实验要求设计一个算法，找到最小的顶点集合S，使得T/S为一个d森林。

2. 问题分析

在本问题中，我们需要设计一个算法，找到最小的顶点集合S，使得T/S为一个d森林。本质上，我们需要从树T中删除一些顶点，使得剩余的顶点构成一个d森林，且删除的顶点数最少。显然，如果一个顶点的子树中的最长路径长度超过了d，那么这个顶点必须被删除，否则这个子树中的某个叶子节点到根节点的路径长度就会超过d。如果一个顶点的子树中的最长路径长度不超过d，则这个顶点可以被保留，但是它的父节点可能需要被删除，因为如果删除这个父节点可以让子树中的某个顶点到根节点的路径长度更短。因此，我们需要从叶子节点开始，向上搜索，并删除需要删除的节点。

3. 算法设计

在本算法中，我们将从叶子节点开始，向上搜索，并删除需要删除的节点，直到剩下的节点构成一个d森林。具体算法流程如下：

初始化marked和markedson数组，分别用于标记已访问的节点和子节点已经访问的个数。

1、在一个循环中，找到未被标记的叶子节点。

2、对于每个叶子节点，如果它的子树中的最长路径长度超过了d，则将其删除，3、否则保留该节点，并更新其父节点的信息。

4、如果节点被删除，则将其添加到S中。

5、将已访问的节点标记为已访问，并更新父节点的子节点已访问个数。

6、重复2~5，直到所有节点都被访问过。

输出S中节点的个数。

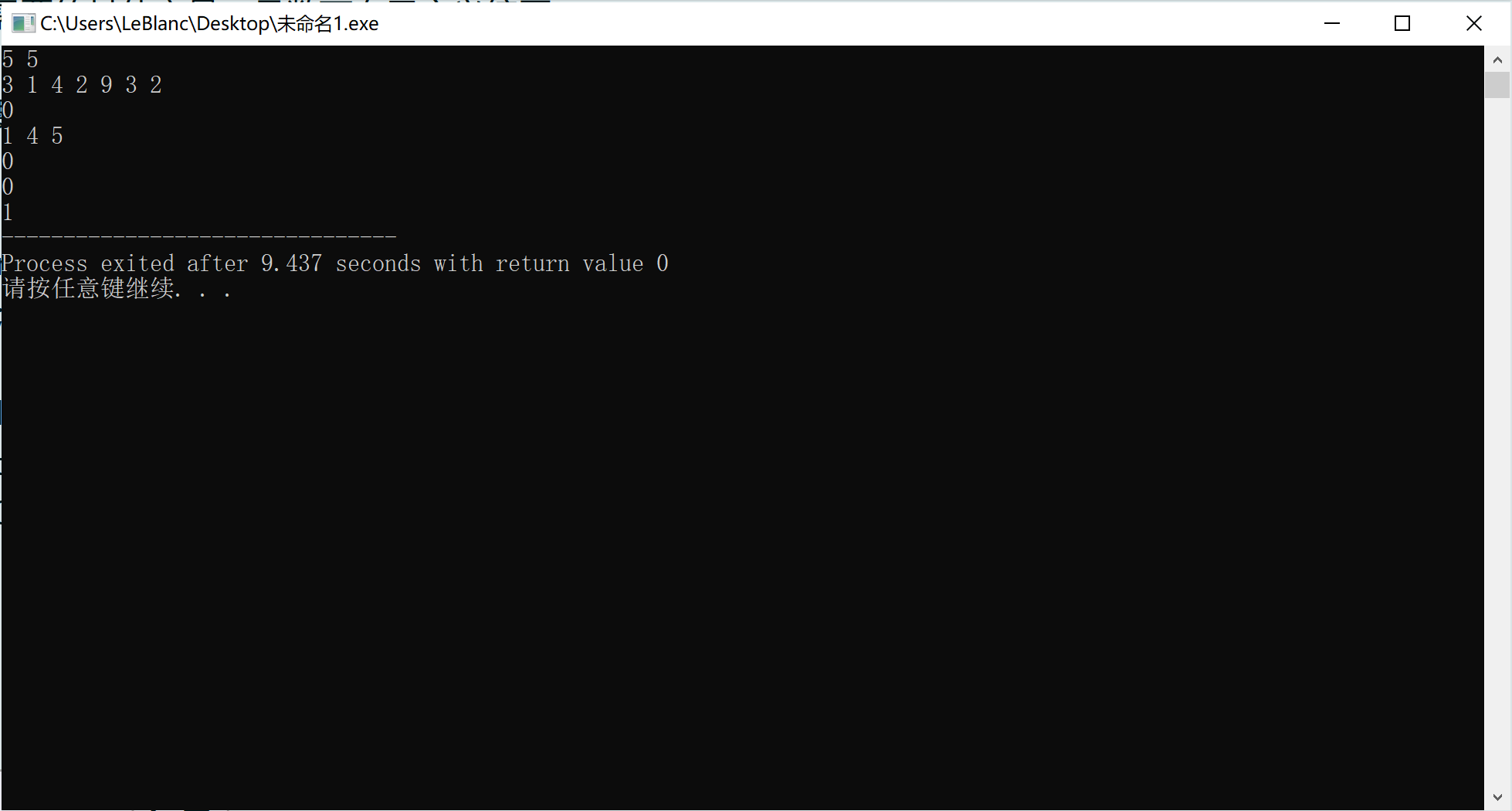
4、算法实现：

我们首先定义了一个vector数组a，用于存储树中的每个节点的信息。其中，a[i].id表示该节点的编号，a[i].son表示该节点的儿子节点的编号，a[i].parent表示该节点的父节点的编号，a[i].ownprice表示从该节点的父节点到该节点的代价，a[i].sonprice表示该节点的所有儿子节点到该节点的代价的最大值。在构造函数中，我们首先输入节点的个数n和路径长度d，然后输入每个节点的儿子节点的个数和与父节点的距离，并构建整个树。

在solution函数中，我们首先定义了两个vector数组，marked和markedson，分别用于标记节点是否被访问和儿子节点已经访问了多少个。然后我们使用一个while循环来不断遍历树，直到所有节点都被访问过为止。在遍历树的过程中，我们首先找到所有没有被标记的节点，并且其所有儿子节点都已经被访问过了。对于这样的节点，我们判断其所有子节点到其最大的路径长度是否小于等于d，如果是，则将其标记为已经访问过，并且更新父节点的sonprice。如果不是，则将该节点加入到S中。

当所有节点都被访问过后，我们输出S中节点的个数，即为最小的顶点集合S。这样，我们就可以得到一个d森林，使得其所有树从根节点到叶子节点的路径长度都不超过d。

5、运行结果



6、总结

本次实验中，我们学习了贪心算法的一种应用，即如何寻找一个d森林，使得其所有树从根节点到叶子节点的路径长度都不超过d。具体地，我们使用了一种自底向上的贪心策略，即先删除叶子节点，再向上删除更高的节点。我们还实现了一个基于vector数组的树数据结构，并通过广度优先搜索构建了整个树。最终，我们通过对所有节点的遍历，找到了一个最小的顶点集合S，使得删除这些节点后，得到的d森林满足条件。

7、代码附录

#include<queue>

#include <iostream>

#include <fstream>

#include <string>

#include <cmath>

#include <vector>

#include <algorithm>

using namespace std;

typedef struct

{

int id;

vector<int>son;

int parent;

int ownprice;//ownprice为从父节点到该处需要的代价

int sonprice;

}Node;

vector<Node>a;

class dTree{

int n1;

int d1;

public:

dTree(int n, int d) {

int i;

Node xx;

for(i=0;i<n;i++)

{

a.push\_back(xx);

a[i].id=i;

a[i].sonprice=0;

}

a[0].ownprice=0;

n1=n;

d1=d;

queue<int>q;

q.push(0);

while(!q.empty())

{ int t=q.front();

q.pop();

int k,j,ds1;//j为新增的点的序号，ds为新增点与父节点之间的带权边长

cin>>k;

for(i=0;i<k;i++)

{cin>>j>>ds1;

a[t].son.push\_back(a[j].id);

a[j].parent=a[t].id;

a[j].ownprice=ds1;

q.push(j);

}

}

//此处为构造函数，初始化构建树。

}

void solution()

{

vector<int>marked(n1,0);//用来做标记节点是否被访问;

vector<int>markedson(n1,0);//儿子节点已经访问了多少个

vector<Node>S;//用来存放被删除的节点;

int ee=0;//用来计数访问过了已经访问过了多少个节点

while(ee<n1)

{ int i;

for(i=0;i<n1;i++)

if((int)a[i].son.size()==markedson[i]&&marked[i]==0)

{

if(a[i].sonprice<=d1)

{

if(i!=0)

{

if(a[a[i].parent].sonprice<a[i].sonprice+a[i].ownprice)

a[a[i].parent].sonprice=a[i].sonprice+a[i].ownprice;

}

}

else

{

S.push\_back(a[i]);

}

ee++;

marked[i]=1;

if(i!=0)

markedson[a[i].parent]+=1;

}

}

cout<<S.size();

}

};

int main() {

int n, d; //n为顶点个数，d为路径长度

cin >> n >> d;

dTree dt(n, d); //构建与初始化树

dt.solution(); //通过solution函数输出结果

return 0;

}