Bonus-1 Path of Equal Weight

Contents

1. Introduction

1.1 Description

We are suppose to mo

2.1 Algorithm Specification

In this algorithm, I used the matrix to store the tree information. I also used an array `parent[]` to store the parent for each node. To find the equal weight path, I used depth first search. Since the sequence got from the DFS is the leaf node, while the output sequence requires to start from the root, so I use stack to store the path from the searched leaf node to the root, and the popping sequence is one sequence of the anwer.

To output the sequences in non-increasing order, I use the insertion sort.

The whole pseudocode is as follows. Note that `N` is the number of nodes, `M` is the number of the non-leaf nodes, and `S` is the given weight.

```
if M==0:
    if weight[0]==S:
        print(weight[0])
        return 0
for each i from 9 to N-1:
    sum[i] <- weight[i]

for each i from 0 to M-1:
    read in the id and k
    childNum[id] <- k
    for each j from 0 to k-1:
        read in the childID
        parent[childID] <- id
        treeMap[id][j] <- childID

// using DFS to search the equal weight
DFS(0)

sort the sequences
output the sequences
```

- The DFS function is as follows

```
void DFS (currentID):

nextChild<-0, i<-0
for each child of currentID:
    sum[child] <- sum[currentID] + weight[child]

    if sum[child]==S:
        if child has no children:
            store the nodes from child to root in a stack
            store the popped sequence
    if sum[child]<S:
        DFS(chlid)

```

3. Testing Results

- **Test point 1: one node only**

> **Input :**
>
> 1 0 5
>
> 5

> **Output :**
>
> 5

- **Test point 2: Max Size N**

> **input**
>
> see the file "Max_N.txt"

> **output**
>
> 1 1 2
> 1 1 1 1

4. Analysis and Comments

4.1 Time Complexity

The time complexity for this algorithm is $max\{O(N), O(k_i M)\}$, where N is the number of nodes, and k_i is the number of children of node i, M is the number of non-leaf nodes. Since the DFS traverse all the nodes in the tree, the read in the tree takes $O(k_i M)$ time. So the time complexity of this algorithm is $max\{O(N), O(k_i M)\}$.

4.2 Space Complexity

The worst case space complexity and for this algorithm is $max\{O(N^2), O(bm)\}$, where N is the number of Nodes, while b and m are the branching factor and the depth of thhe tree respectively. Since in the code, I use the matrix to store the information of the tree. And the space complexity of the DFS is $O(bm)$.

5. Appendix

> Source Code in C

```
#include<stdio.h>
#include<stdlib.h>
```

```c

#define MAX 101     // the maximum number of nodes

int N, M, S;
int weight[MAX], parent[MAX] = { -1 }, sum[MAX], treeMap[MAX][MAX], childNumber[MAX], result[MAX][MAX];
int stack[MAX], stackPointer = -1, resultPointer = -1;
/*************************************************************************
                    Variables Used

* N: The number of nodes in the tree.

* M: The number of non-leaf nodes in the tree

* S: The given weight number we are going to search

* weight[]: An array that stores the weight of each node

* parent[]: An array that stores the parent of each node (used in storing
            the result)

* sum[]: An array that stores the sum of weights from the root to each node

* treeMap[][]: A matrix that store the tree. treeMap[i][] is an array that
            stores all the children of node i.

* childNumber[]: An array that stores the number of children of each node.

* result[]: An array that stores the result.

* stack[]: A stack that stores each sequence of result (to get the reverse order)

* stackPointer: the pointer of stack[].

* resultPointer: the pointer of result[].
*************************************************************************/


void depthFirstSearch(int currentId);
void push(int* stack, int val);
void pop(int* stack);
int top(int* stack);
int isEmpty(int* stack);
void swap(int* arr1, int* arr2);
int check(int* array1, int* array2);
void storeResult(int id);
void printResult(void);

/*********************************************************
                Functions

* depthFirstSearch: Using depth first search to find the
            equal paths.

* push; pop; top; isEmpty: Basic operations in stack.

```

```
* check: To determine two arrays which is larger.

* swap: Swap the two arrays (exchange all the elements)

* storeResult: store one sequence of result.

* printResult: Sort the result array and output the
            results in specific format.

*********************************************************/


int main()
{
    int i = 0, j = 0;
    int id = 0, k = 0, childId = 0;

    scanf("%d %d %d", &N, &M, &S);      // read in the N,M,S

    for (i = 0; i < N; i++)             //read in the weight of each node
        scanf("%d", &weight[i]);

    if (M == 0) {                       // there's only one node
        if (weight[0] == S) printf("%d\n", weight[0]);
        return 0;
    }

    for (i = 0; i < N; i++)             // initialize the sum[] array
        sum[i] = weight[i];

    for (i = 0; i < M; i++) {
        scanf("%d %d", &id, &k);        // read in the id and the number of children
        childNumber[id] = k;
        for (j = 0; j < k; j++) {
            scanf("%d", &childId);
            parent[childId] = id;       // initialize the parent[] array
            treeMap[id][j] = childId;   // initialize the treeMap[][] matrix
        }

    }

    depthFirstSearch(0);    // using depth first search to find the equal path, starting from the root

    printResult();          // output the result in specific format

    return 0;
}

void depthFirstSearch(int currentId)
{
    int nextChild = 0, i = 0;
    for (i = 0; i < childNumber[currentId]; i++) {              // traverse all the children of current node
        nextChild = treeMap[currentId][i];                      // get the child node id
```

```
        sum[nextChild] = sum[currentId] + weight[nextChild];    // update the sum[nextChild]

        if (sum[nextChild] == S) {          // if the path weight is equal to S
            if (childNumber[nextChild] != 0)    // not a leaf node, ignore
                continue;
            else  storeResult(nextChild);       // a leaf node, store result
        }
        else if (sum[nextChild] < S)            // if the sum is less than S, start searching next level
            depthFirstSearch(nextChild);
    }
}

void push(int* stack, int val)
{
    stack[++stackPointer] = val;
}

void pop(int* stack)
{
    stackPointer--;
}

int top(int* stack)
{
    return stack[stackPointer];
}

int isEmpty(int* stack)
{
    return stackPointer == -1;
}

// since the DFS find the leaf node, we need to store the result starting from the root
void storeResult(int id)
{
    int i = 0;
    while (id != -1) {      // push the path to the stack
        push(stack, weight[id]);
        id = parent[id];
    }
    resultPointer++;
    while (!isEmpty(stack)) {
        result[resultPointer][i++] = top(stack);  // store the path nodes from root to leaf
        pop(stack);
    }
}

void swap(int* arr1, int* arr2)
{
    int i = 0;
    int* temp = (int*)malloc(MAX * sizeof(int));
    for (i = 0; i < MAX; i++) {
```

```c
        temp[i] = arr1[i];
        arr1[i] = arr2[i];
        arr2[i] = temp[i];
    }
    free(temp);
}

int check(int* array1, int* array2)
{
    int i = 0;
    for (i = 0; i < MAX; i++) {
        if (array1[i] == array2[i])
            continue;
        else if (array1[i] > array2[i])
            return 1;
        else return -1;
    }
    return 0;
}

void printResult(void) // sort the result sequence by insertion sort
{
    int i = 0, j = 0, k = 0;
    int* temp = (int*)malloc(sizeof(int) * MAX), * key =
(int*)malloc(sizeof(int) * MAX);
    for (i = 1; i <= resultPointer; i++) {
        for (k = 0; k < MAX; k++)
            key[k] = result[i][k];
        j = i - 1;
        while (j >= 0 && check(key, result[j]) == 1) {
            swap(result[j + 1], result[j]);
            j--;
        }
        for (k = 0; k < MAX; k++)
            result[j + 1][k] = key[k];
    }
    for (i = 0; i <= resultPointer; i++) {
        printf("%d", result[i][0]);
        j = 1;
        while (result[i][j] != 0) {
            printf(" %d", result[i][j++]);
        }
        printf("\n");
    }
}
```