【BFS】-BOSS的收入

题目描述与示例

题目描述

一个 `XX` 产品行销总公司，只有一个boss，其有若干一级分销，一级分销又有若干二级分销，每个分销只有唯一的上级分销。规定，每个月，下级分销需要将自己的总收入(自己的+下级上交的)每满 `100` 元上交 `15` 元给自己的上级。

现给出一组分销的关系，和每个分销的收入，请找出boss并计算出这个boss的收入。

比如:

收入 `100` 元，上交 `15` 元;

收入 `199` 元（ `99` 元不够 `100` ），上交 `15` 元，

收入 `200` 元，上交 `30` 元。

输入：

分销关系和收入: `[[分销id 上级分销的Id 收入]，[分销id 上级分销的id 收入]，[分销id 上级分销的id 收入]]`

分销ID范围 `0..65535`

收入范围 `0..65535` ，单位元

提示：输入的数据只存在 `1` 个boss，不存在环路

输出： `[boss的ID，总收入]`

输入描述

第 `1` 行输入关系的总数量 `N`

第 `2` 行开始，输入关系信息，格式： `分销ID 上级分销ID 收入`

比如:

```
5
```

```
1 0 100
```

```
2 0 199
```

```
3 0 200
```

`4 0 200`

`5 0 200`

输出描述

输出：`boss的ID 总收入`

比如：

`0 120`

补充说明

给定的输入数据都是合法的，不存在环路，重复的

示例

输入

```
5
1 0 100
2 0 199
3 0 200
4 0 200
5 0 200
```

输出

```
0 120
```

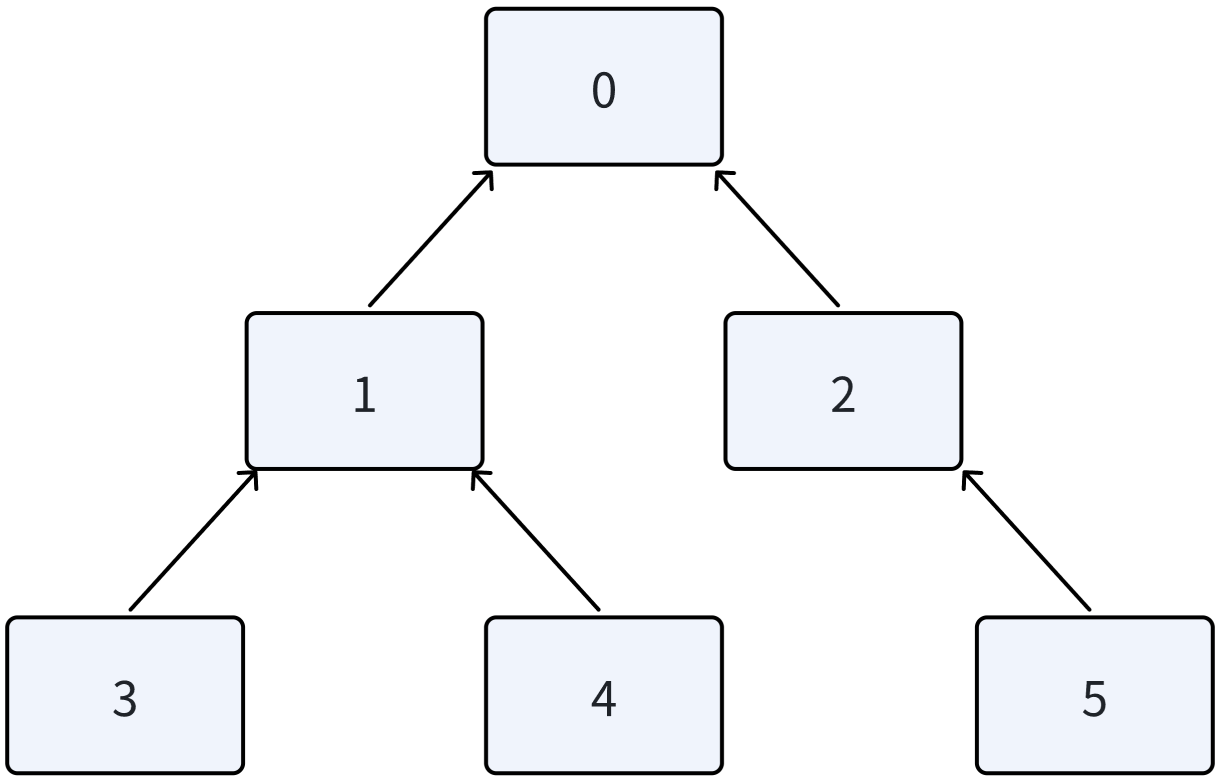
解题思路

拓扑排序BFS解法

很明显这个层层分销的制度，可以**使用树形结构**来表示。譬如对于例子

```
5
1 0 100
2 0 199
3 1 200
4 1 200
5 2 200
```

可以画成如下树形结构



每一个上级的收入不仅取决于他自己的收入，还取决于其直接下属的收入。

以这个例子为例，如果我们想计算节点 `0` 从节点 `1` 得到多少收入，就必须先计算节点 `1` 的总收入。而如果想计算节点 `1` 的总收入，又必须先计算节点 `1` 从节点 `3` 和节点 `4` 分别获得多少收入。

很显然这存在依赖关系：**我们必须先把下层节点的总收入计算完之后，才能将总收入进行抽成，来计算当前节点的的总收入**。

对于这种存在依赖的问题，我们可以使用**拓扑排序**来完成。直接套模板即可完成。

注意本题并没有直接告知根节点的ID，因此需要找到唯一的非子节点来作为根节点。

*自底向上的DFS解法

当然，熟悉树和递归的同学，也容易想到能够用**自底向上的DFS**来完成这个题目。

本文不做赘述，后面提供包含详尽注释的代码。

代码

解法一：拓扑排序BFS

Python

```python
# 题目：【BFS】2024E-BOSS的收入
# 分值：200
# 作者：许老师-闭着眼睛学数理化
# 算法：BFS/拓扑排序
# 代码看不懂的地方，请直接在群上提问

from collections import defaultdict, deque

```

```python
# 输入边的个数
n = int(input())

# 构建邻接表，key是子节点，value是父节点
# 由于每一个节点最多只有一个父节点
# 所以parents邻接表的value无需设置默认值为list
parents = dict()

# 构建总收入哈希表，key是节点名，value是该节点的收入
total_money = defaultdict(int)

# 构建入度哈希表，key是节点名，value是入度
indegree = defaultdict(int)


# 循环n行，输入n行
for _ in range(n):
    # 在实际考试中，发现必须加入这里的try-except语句才能够满分
    # 题目的输入存在一些未知的错误，不加上只能够通过95%的用例
    try:
        # 输入子节点c，父节点p，子节点最初的收入money
        # c表示children，p表示parent
        c, p, money = map(int, input().split())
        # 在parents邻接表中储存c的父节点为p
        parents[c] = p
        # 在总收入哈希表中初始化c的收入，记录为money
        total_money[c] = money
        # 后续需要进行拓扑排序，父节点p的入度+1
        indegree[p] += 1
    except:
        break


# 寻找boss根节点root，根节点存在以下特征：
# 1. 入度不为0（位于indegree的key中）
# 2. 不是任何一个节点的子节点（不位于parents的key中）
for node in indegree.keys():
    if node not in parents:
        root = node
        break

# 构建队列q维护拓扑排序过程
q = deque()
# 所有入度为0的节点，都是初始的叶节点，存入队列q中
for c in parents.keys():
    if indegree[c] == 0:
        q.append(c)
```

```python

# 拓扑排序过程
while q:
    # 弹出队头元素，为当前节点c
    c = q.popleft()
    # 如果遍历到根节点root，则直接退出循环
    if c == root:
        break
    # 获得当前节点c的父节点p
    p = parents[c]
    # 父节点的入度+1
    indegree[p] -= 1
    # 弹出的当前节点c的收入已经计算完毕
    # 将其收入整除100后乘15，是提供给父节点p的分销佣金
    # 将该分销佣金加入父节点p的收入中
    total_money[p] += total_money[c] // 100 * 15
    # 若此时父节点的入度为0，则说明其所有子节点均已考虑
    # 该父节点p的总收入计算完毕，将其加入队列
    if indegree[p] == 0:
        q.append(p)

# 输出root的id以及其收入total_money[root]
print(f"{root} {total_money[root]}")
```

Java

```java
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();

        // 构建邻接表，key是子节点，value是父节点
        Map<Integer, Integer> parents = new HashMap<>();

        // 构建总收入哈希表，key是节点名，value是该节点的收入
        Map<Integer, Integer> totalMoney = new HashMap<>();

        // 构建入度哈希表，key是节点名，value是入度
        Map<Integer, Integer> indegree = new HashMap<>();

        // 循环n行，输入n行
        for (int i = 0; i < n; i++) {
```

```
            try {
                // 输入子节点c，父节点p，子节点最初的收入money
                int c = scanner.nextInt();
                int p = scanner.nextInt();
                int money = scanner.nextInt();

                // 在parents邻接表中储存c的父节点为p
                parents.put(c, p);

                // 在总收入哈希表中初始化c的收入，记录为money
                totalMoney.put(c, money);

                // 后续需要进行拓扑排序，父节点p的入度+1
                indegree.put(p, indegree.getOrDefault(p, 0) + 1);

            } catch (Exception e) {
                break;
            }
        }

        // 寻找boss根节点root，根节点存在以下特征：
        // 1. 入度不为0（位于indegree的key中）
        // 2. 不是任何一个节点的子节点（不位于parents的key中）
        int root = -1;
        for (int node : indegree.keySet()) {
            if (!parents.containsKey(node)) {
                root = node;
                break;
            }
        }

        // 构建队列q维护拓扑排序过程
        Queue<Integer> q = new LinkedList<>();
        // 所有入度为0的节点，都是初始的叶节点，存入队列q中
        for (int c : parents.keySet()) {
            if (indegree.getOrDefault(c, 0) == 0) {
                q.offer(c);
            }
        }

        // 拓扑排序过程
        while (!q.isEmpty()) {
            // 弹出队头元素，为当前节点c
            int c = q.poll();

            // 如果遍历到根节点root，则直接退出循环
            if (c == root) {
```

```
                break;
            }

            // 获得当前节点c的父节点p
            int p = parents.get(c);

            // 父节点的入度-1
            indegree.put(p, indegree.get(p) - 1);

            // 弹出的当前节点c的收入已经计算完毕
            // 将其收入整除100后乘15，是提供给父节点p的分销佣金
            // 将该分销佣金加入父节点p的收入中
            totalMoney.put(p, totalMoney.getOrDefault(p, 0) +
totalMoney.get(c) / 100 * 15);

            // 若此时父节点的入度为0，则说明其所有子节点均已考虑
            // 该父节点p的总收入计算完毕，将其加入队列
            if (indegree.get(p) == 0) {
                q.offer(p);
            }
        }

        // 输出root的id以及其收入totalMoney.get(root)
        System.out.println(root + " " + totalMoney.get(root));

        scanner.close();
    }
}

```

C++

```
#include <iostream>
#include <unordered_map>
#include <queue>

using namespace std;

int main() {
    int n;
    cin >> n;

    // 构建邻接表，key是子节点，value是父节点
    unordered_map<int, int> parents;

```

```cpp
    // 构建总收入哈希表，key是节点名，value是该节点的收入
    unordered_map<int, int> total_money;

    // 构建入度哈希表，key是节点名，value是入度
    unordered_map<int, int> indegree;

    // 循环n行，输入n行
    for (int i = 0; i < n; i++) {
        // 在实际考试中，发现必须加入这里的try-catch语句才能够满分
        // 题目的输入存在一些未知的错误，不加上只能够通过95%的用例
        try {
            // 输入子节点c，父节点p，子节点最初的收入money
            int c, p, money;
            cin >> c >> p >> money;

            // 在parents邻接表中储存c的父节点为p
            parents[c] = p;

            // 在总收入哈希表中初始化c的收入，记录为money
            total_money[c] = money;

            // 后续需要进行拓扑排序，父节点p的入度+1
            indegree[p]++;
        } catch (...) {
            break;
        }
    }

    // 寻找boss根节点root，根节点存在以下特征：
    // 1. 入度不为0（位于indegree的key中）
    // 2. 不是任何一个节点的子节点（不位于parents的key中）
    int root = -1;
    for (const auto& pair : indegree) {
        int node = pair.first;
        if (parents.find(node) == parents.end()) {
            root = node;
            break;
        }
    }

    // 构建队列q维护拓扑排序过程
    queue<int> q;
    // 所有入度为0的节点，都是初始的叶节点，存入队列q中
    for (const auto& pair : parents) {
        int c = pair.first;
        if (indegree[c] == 0) {
            q.push(c);
```

```
        }
    }

    // 拓扑排序过程
    while (!q.empty()) {
        // 弹出队头元素，为当前节点c
        int c = q.front();
        q.pop();

        if (c == root)
            break;

        // 获得当前节点c的父节点p
        int p = parents[c];

        // 父节点的入度-1
        indegree[p]--;

        // 弹出的当前节点c的收入已经计算完毕
        // 将其收入整除100后乘15，是提供给父节点p的分销佣金
        // 将该分销佣金加入父节点p的收入中
        total_money[p] += total_money[c] / 100 * 15;

        // 若此时父节点的入度为0，则说明其所有子节点均已考虑
        // 该父节点p的总收入计算完毕，将其加入队列
        if (indegree[p] == 0) {
            q.push(p);
        }
    }

    // 输出root的id以及其收入total_money[root]
    cout << root << " " << total_money[root] << endl;

    return 0;
}

```

时空复杂度

时间复杂度：`O(N)`。需要遍历每一个节点。

空间复杂度：`O(N)`。入度哈希表和邻接表所占空间。

*解法二：自底向上的DFS

Python

```python
# 题目：【BFS】2024E-BOSS的收入
# 分值：200
# 作者：许老师-闭着眼睛学数理化
# 算法：自底向上DFS
# 代码看不懂的地方，请直接在群上提问

from collections import defaultdict


# 自底向上的dfs函数
# node为当前节点
# neighbor_dic为邻接表
# total_money为每一个节点的总收入
def dfs(node, neighbor_dic, total_money):
    # 如果当前节点node不是一个父节点，则说明其为叶子节点
    # 叶节点的总收入无需修改
    # 直接返回
    if node not in neighbor_dic:
        return

    # 考虑当前节点的所有子节点child
    for child in neighbor_dic[node]:
        # 自底向上，先对子节点进行DFS调用，更新子节点的总收入
        dfs(child, neighbor_dic, total_money)
        # 子节点的DFS调用后，子节点的总收入已经计算完毕
        # 将其更新入当前节点中
        total_money[node] += total_money[child] // 100 * 15

    return

# 输入边的个数
n = int(input())

# 构建表示树形结构的邻接表
# 其中key是节点名，value是其所有子节点children构成的列表
neighbor_dic = defaultdict(list)

# 构建总收入哈希表，key是节点名，value是该节点的收入
total_money = defaultdict(int)

```

```python

# 循环n行，输入n行
for _ in range(n):
    # 在实际考试中，发现必须加入这里的try-except语句才能够满分
    # 题目的输入存在一些未知的错误，不加上只能够通过95%的用例
    try:
        # 输入子节点c，父节点p，子节点最初的收入money
        # c表示children，p表示parent
        c, p, money = map(int, input().split())
        # 在邻接表中储存节点p的子节点包含c
        neighbor_dic[p].append(c)
        # 在总收入哈希表中初始化c的收入，记录为money
        total_money[c] = money
    except:
        break


# 寻找根节点，存在于neighbor_dic中，
# 且未不位于total_money中的节点为根节点
for p in neighbor_dic:
    if p not in total_money:
        root = p
        break

# 递归入口，调用根节点root
dfs(root, neighbor_dic, total_money)

# 输出root的id以及其收入total_money[root]
print(f"{root} {total_money[root]}")
```

Java

```java
import java.util.*;

public class Main {

    // 自底向上的DFS函数
    // node为当前节点
    // neighborDic为邻接表
    // totalMoney为每一个节点的总收入
    public static void dfs(int node, Map<Integer, List<Integer>> neighborDic,
Map<Integer, Integer> totalMoney) {
        // 如果当前节点node不是一个父节点，则说明其为叶子节点
        // 叶节点的总收入无需修改，直接返回
        if (!neighborDic.containsKey(node)) {
```

```
            return;
        }

        // 考虑当前节点的所有子节点child
        for (int child : neighborDic.get(node)) {
            // 自底向上，先对子节点进行DFS调用，更新子节点的总收入
            dfs(child, neighborDic, totalMoney);
            // 子节点的DFS调用后，子节点的总收入已经计算完毕
            // 将其更新入当前节点中
            totalMoney.put(node, totalMoney.getOrDefault(node, 0) +
totalMoney.get(child) / 100 * 15);
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();

        // 构建表示树形结构的邻接表
        Map<Integer, List<Integer>> neighborDic = new HashMap<>();

        // 构建总收入哈希表，key是节点名，value是该节点的收入
        Map<Integer, Integer> totalMoney = new HashMap<>();

        // 循环n行，输入n行
        for (int i = 0; i < n; i++) {
            try {
                // 输入子节点c，父节点p，子节点最初的收入money
                int c = scanner.nextInt();
                int p = scanner.nextInt();
                int money = scanner.nextInt();

                // 在邻接表中储存节点p的子节点包含c
                neighborDic.computeIfAbsent(p, k -> new ArrayList<>()).add(c);

                // 在总收入哈希表中初始化c的收入，记录为money
                totalMoney.put(c, money);
            } catch (Exception e) {
                break;
            }
        }

        // 寻找根节点，存在于neighborDic中，且未不位于totalMoney中的节点为根节点
        int root = -1;
        for (int p : neighborDic.keySet()) {
            if (!totalMoney.containsKey(p)) {
                root = p;
```

```
                break;
            }
        }

        // 递归入口，调用根节点root
        dfs(root, neighborDic, totalMoney);

        // 输出root的id以及其收入totalMoney.get(root)
        System.out.println(root + " " + totalMoney.get(root));

        scanner.close();
    }
}

```

C++

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <queue>

using namespace std;

// 自底向上的DFS函数
// node为当前节点
// neighborDic为邻接表
// totalMoney为每一个节点的总收入
void dfs(int node, unordered_map<int, vector<int>>& neighborDic,
unordered_map<int, int>& totalMoney) {
    // 如果当前节点node不是一个父节点，则说明其为叶子节点
    // 叶节点的总收入无需修改，直接返回
    if (neighborDic.find(node) == neighborDic.end()) {
        return;
    }

    // 考虑当前节点的所有子节点child
    for (int child : neighborDic[node]) {
        // 自底向上，先对子节点进行DFS调用，更新子节点的总收入
        dfs(child, neighborDic, totalMoney);
        // 子节点的DFS调用后，子节点的总收入已经计算完毕
        // 将其更新入当前节点中
        totalMoney[node] += totalMoney[child] / 100 * 15;
    }
}
```

```cpp

int main() {
    int n;
    cin >> n;

    // 构建表示树形结构的邻接表
    unordered_map<int, vector<int>> neighborDic;

    // 构建总收入哈希表，key是节点名，value是该节点的收入
    unordered_map<int, int> totalMoney;

    // 循环n行，输入n行
    for (int i = 0; i < n; i++) {
        try {
            // 输入子节点c，父节点p，子节点最初的收入money
            int c, p, money;
            cin >> c >> p >> money;

            // 在邻接表中储存节点p的子节点包含c
            neighborDic[p].push_back(c);

            // 在总收入哈希表中初始化c的收入，记录为money
            totalMoney[c] = money;
        } catch (...) {
            break;
        }
    }

    // 寻找根节点，存在于neighborDic中，且未不位于totalMoney中的节点为根节点
    int root = -1;
    for (const auto& pair : neighborDic) {
        int p = pair.first;
        if (totalMoney.find(p) == totalMoney.end()) {
            root = p;
            break;
        }
    }

    // 递归入口，调用根节点root
    dfs(root, neighborDic, totalMoney);

    // 输出root的id以及其收入totalMoney[root]
    cout << root << " " << totalMoney[root] << endl;

    return 0;
}
```

时空复杂度

时间复杂度：`O(N)`。需要遍历每一个节点。

空间复杂度：`O(N)`。邻接表所占空间。