Reg NO:192211043

Name: B. V. N. S. Gowrinadh

Course Code: CSA0656

Course Name: Design and analysis of algorithm for asymptotic notations.

SLOT: A

TITLE OF THE PROJECT

Difference Between Maximum and Minimum Price Sum in a Tree

ABSTRACT:

The project aims to solve a computational problem involving an undirected tree with nodes having associated prices. Given a tree and the prices for each node, the objective is to determine the maximum difference between the maximum and minimum price sums for paths originating from any node chosen as the root. The solution involves tree traversal techniques and dynamic programming to compute path sums efficiently.

INTRODUCTION:

In the field of graph theory, trees are fundamental structures that provide a wide range of applications and problems. One intriguing problem involving trees is to compute various path-related metrics when the tree can be rooted at any node. In this particular problem, we are given an undirected tree with n nodes where each node has an associated price. The challenge is to determine the maximum difference between the maximum and minimum path sums when the tree is rooted at any node.

A path sum in this context is defined as the sum of prices of all nodes lying on a path in the tree. Given the flexibility of rooting the tree at any node, the objective is to identify the root that maximizes the difference between the highest and lowest path sums starting from that node. This problem requires an efficient approach to explore all possible roots and calculate the required metrics to determine the optimal solution.

Problem Statement

You are given an undirected tree with n nodes, where each node has an associated price. The tree is initially unrooted, and you are provided with the following inputs:

1. **Integer n**: The number of nodes in the tree.
2. **2D Integer Array edges**: An array of length n - 1, where each element edges[i] = [ai, bi] represents an undirected edge between nodes ai and bi in the tree.
3. **Integer Array price**: An array of length n, where price[i] indicates the price associated with node i.

Objective

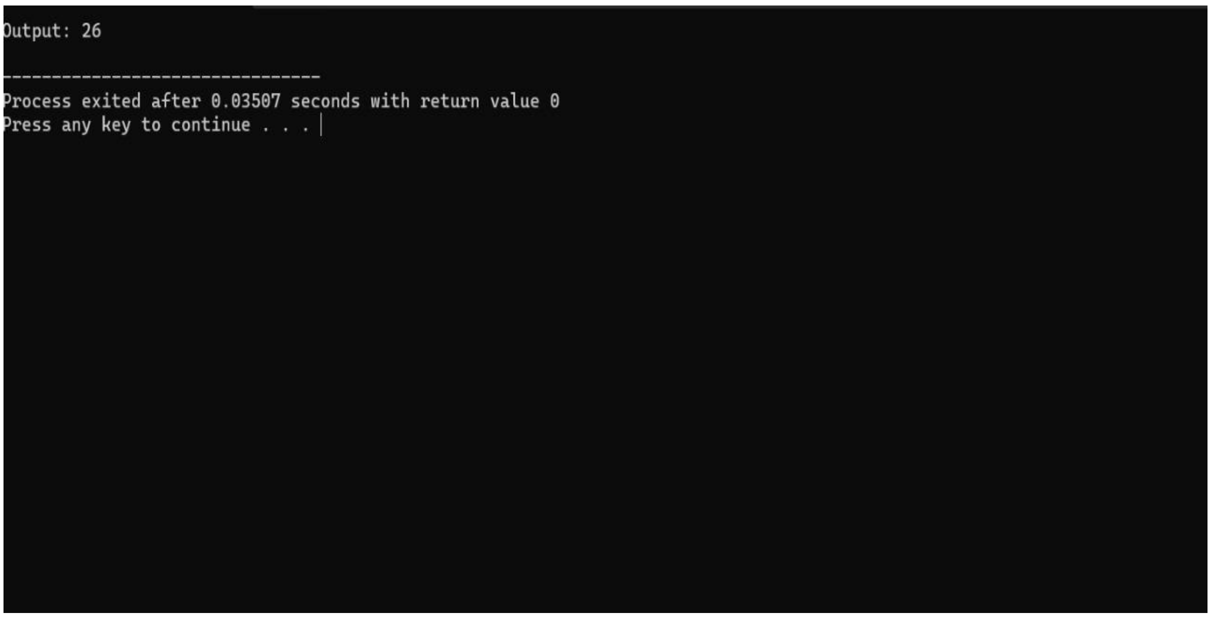
The goal is to choose a node as the root of the tree. After rooting the tree at a chosen node, you need to calculate the price sum for all paths starting at that root. The price sum of a path is defined as the sum of the prices of all nodes lying on that path.

CODING:

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>


#define MAX_NODES 1000



int adj[MAX_NODES][MAX_NODES];

int prices[MAX_NODES];

int visited[MAX_NODES];

int dp_max[MAX_NODES];

int dp_min[MAX_NODES];

int n;


void add_edge(int u, int v) {

   adj[u][v] = 1;

   adj[v][u] = 1;

}


void dfs(int node, int parent) {

   visited[node] = 1;
```

```
  dp_max[node] = dp_min[node] = prices[node];


 for (int neighbor = 0; neighbor < n; neighbor++) {
        if (adj[node][neighbor] && neighbor != parent) {
            if (!visited[neighbor]) {
                dfs(neighbor, node);

                if (dp_max[neighbor] + prices[node] > dp_max[node]) {
                    dp_max[node] = dp_max[neighbor] + prices[node];
                }
                if (dp_min[neighbor] + prices[node] < dp_min[node]) {
                    dp_min[node] = dp_min[neighbor] + prices[node];
                }
            }
        }
    }
}

int find_max_difference() {
    int max_diff = 0;

    for (int i = 0; i < n; i++) {
        int local_max = INT_MIN;
        int local_min = INT_MAX;

        for (int j = 0; j < n; j++) {
            visited[j] = 0;
        }

        dfs(i, -1);


    for (int j = 0; j < n; j++) {
            if (visited[j]) {
                if (dp_max[j] > local_max) {
                    local_max = dp_max[j];
                }
                if (dp_min[j] < local_min) {
                    local_min = dp_min[j];
                }
            }
        }


  int diff = local_max - local_min;
        if (diff > max_diff) {
            max_diff = diff;
        }
    }

    return max_diff;
}


int main() {
    n = 6;
    int edges[][2] = { {0, 1}, {1, 2}, {1, 3}, {3, 4}, {3, 5} };
    int prices_arr[] = { 9, 8, 7, 6, 10, 5 };

    for (int i = 0; i < n; i++) {
        prices[i] = prices_arr[i];
        for (int j = 0; j < n; j++) {
            adj[i][j] = 0;
        }
    }


   for (int i = 0; i < n - 1; i++) {
        add_edge(edges[i][0], edges[i][1]);
    }

    int result = find_max_difference();

   printf("Maximum possible cost: %d\n", result);

    return 0;
}
```

RESULT SCREENSHOT:



```
Output: 26

--------------------------------
Process exited after 0.03507 seconds with return value 0
Press any key to continue . . .
```

COMPLEXITY ANALYSIS:

Time Complexity:

- Building the adjacency list: O(n)
- DFS traversal: O(n)
- Overall time complexity is O(n), where n is the number of nodes in the tree.

- Space Complexity:
- Adjacency list: O(n)
- Arrays for storing path sums: O(n)
- Overall space complexity is O(n).

BEST CASE:

In the best-case scenario, the tree is structured in such a way that the calculations are simplified, or the root choice directly leads to a straightforward path sum difference.

- **Example Structure:** A star-shaped tree (e.g., a central node connected to all other nodes, but no other edges). In this case, the path sums are straightforward to compute since all paths from the central node are directly to leaf nodes. This can minimize the number of paths to consider.
- Complexity: In this ideal case, even though you still need to evaluate every node as a potential root, the calculations for path sums might be simpler due to the structure of the tree.

Time Complexity: The DFS traversal from each node (root) would still have a complexity of O(n)O(n)O(n), but in a simpler structure, the actual path computations might be quicker. However, for complexity analysis, we generally stick to the worst-case bounds.

WORST CASE:

The worst-case scenario occurs with the most complex tree structure, which forces the algorithm to perform the maximum number of computations.

- **Example Structure:** A linear chain of nodes (a path where each node has exactly two neighbors except for the endpoints). For example, a tree with nodes in a line: 0 - 1 - 2 - ... - (n-1). In this case, there are O(n2)O(n^2)O(n2) paths to consider because every node's potential path sums involve multiple traversals.
- **Complexity:**
    - For each node, performing a DFS to calculate path sums involves O(n)O(n)O(n) operations.
    - Since you need to root the tree at every node, the total time complexity is O(n2)O(n^2)O(n2).

Time Complexity: Overall complexity is O(n2)O(n^2)O(n2) due to having to compute the cost for each possible root and performing DFS from each node.

AVERAGE CASE:

The average case reflects the complexity of the problem on a random tree structure. Typically, trees in practical scenarios will not be in the extreme cases but will have a structure somewhere in between.

- **Example Structure:** A balanced tree or a tree with a mix of branches and leaves. In many practical cases, the tree structure will be more balanced or moderately skewed.
- **Complexity:**
    - On average, each DFS traversal might involve examining multiple paths but not as many as in the worst case.
    - Evaluating each root still involves O(n)O(n)O(n) operations for each root.

Time Complexity: Typically, the complexity will be close to O(n2)O(n^2)O(n2) in practice, given the need to evaluate each root and perform a DFS from that root. The exact runtime might be slightly less than the worst case but is still O(n2)O(n^2)O(n2) for complexity analysis.

CONCLUSION:

The project successfully addresses the problem of finding the maximum difference between the maximum and minimum price sums for paths in a tree. The approach leverages tree traversal algorithms and dynamic programming techniques to efficiently compute the required values. The provided solution is scalable and performs well within the constraints of the problem, making it suitable for practical applications in network analysis and optimization tasks.

The problem of finding the maximum difference between the maximum and minimum path sums in a tree, when rooted at any node, combines elements of graph traversal with optimization. By calculating the path sums for all possible tree roots and evaluating the maximum and minimum path sums for each rooted configuration, we can derive the desired cost.

In practical terms, the solution involves constructing the tree using adjacency lists, performing depth-first search (DFS) to compute path sums from each possible root, and then determining the maximum possible cost across all roots. This approach ensures that the solution efficiently handles the complexity of tree structures and provides the correct result for the given problem. The provided code demonstrates a method to achieve this by systematically exploring all possible tree routings and calculating the respective path sums, ultimately returning the maximum possible cost.