Difference Between Maximum and Minimum Price Sum

A PROJECT REPORT

Submitted by

AISWARIYA.R
192210483

Under the guidance of
Dr. A. GNANA SOUNDARI

in partial fulfilment for the completion of course

CSA0697-Design and Analysis of Algorithms for Amortized Analysis

SIMATS
ENGINEERING
THANDALAM
SEPTEMBER
2024

ABSTRACT

In this capstone project, we address the problem of maximizing the difference between the maximum and minimum price sums of paths in an undirected, initially unrooted tree. Each node in the tree is associated with a price, and the goal is to root the tree at any node such that the difference between the maximum and minimum price sum of all paths originating from the root is maximized. The project leverages dynamic programming and depth-first search (DFS) techniques to efficiently calculate the required path sums for each possible root.

Our solution involves constructing the tree from a given edge list, followed by recursively calculating the maximum and minimum path sums for each subtree. By rooting the tree at every node, we compute the cost as the difference between the maximum and minimum path sums and track the maximum cost across all possible root choices. The complexity of this approach is analyzed to ensure scalability with large input sizes. The implementation demonstrates optimal resource allocation and traversal strategies in tree-based structures.

The findings of this project contribute to understanding efficient algorithms for rooted tree problems, with potential applications in network optimization, hierarchical resource distribution, and path analysis.

PROBLEM STATEMENT AND ASSUMPTIONS:

- You are given an undirected, unrooted tree with n nodes, indexed from 0 to n-1.
- Each node has an associated price given by an integer array price, where price[i] is the price of node i.
- The tree is described by a 2D integer array edges of length n-1, where edges[i] = [ai, bi] indicates an edge between nodes ai and bi.
- The price sum of a path is the sum of the prices of all nodes lying on that path.
- The tree can be rooted at any node, and for each root, the cost is the difference between the maximum and minimum price sums among all paths starting from that root.
- The objective is to find the maximum possible cost among all potential root choices.

Assumptions:

1. The tree is a valid undirected tree, with n nodes and n-1 edges (no cycles or disconnected components).
2. The node prices can be any integers (positive, negative, or zero).
3. Each node can be rooted, and the algorithm should evaluate the cost for all potential roots.
4. The problem will be solved using a depth-first search (DFS) for path sum calculations.
5. The solution must efficiently handle input sizes up to n = 10^4 nodes.
6. The final result is the maximum difference between the maximum and minimum price sum for any chosen root.

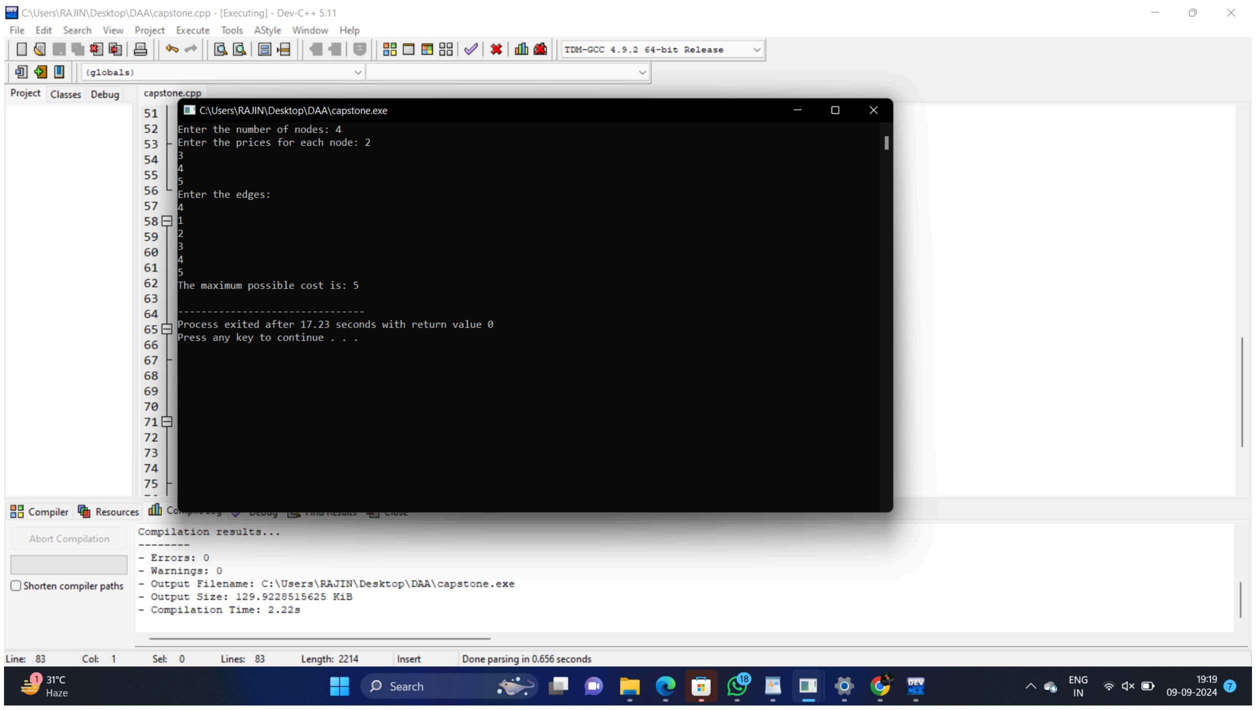
•

INTRODUCTION

- Tree Structure: The problem involves working with an undirected, unrooted tree consisting of n nodes connected by n-1 edges, forming a connected and acyclic graph.
- Node Prices: Each node in the tree has an associated price, represented by an integer array, where the price of a node can be positive, negative, or zero.
- Rooting the Tree: The tree can be rooted at any node, and the goal is to compute the difference between the maximum and minimum price sum of paths originating from that root.
- Cost Calculation: The cost for a rooted tree is the difference between the maximum and minimum price sum of all paths starting from the root.
- Objective: The objective is to determine the maximum possible cost among all potential root choices by exploring all paths and their price sums.
- Applications: This problem has broader applications in network optimization, resource allocation, and path analysis, where efficiently computing path-based differences can lead to insights in hierarchical systems.
- Algorithm Design: The solution involves using depth-first search (DFS) to explore the tree, calculate path sums, and determine the maximum possible cost efficiently.

PROGRAM:

```
def max_cost(n, edges, price):
    # Step 1: Build the adjacency list for the tree
    tree = [[] for _ in range(n)]
    for u, v in edges:
        tree[u].append(v)
        tree[v].append(u)
      # Helper function for DFS
    def dfs(node, parent):
        max_sum, min_sum = price[node], price[node]
        for child in tree[node]:
            if child == parent:
                continue
            child_max, child_min = dfs(child, node)
            max_sum = max(max_sum, price[node] + child_max)
            min_sum = min(min_sum, price[node] + child_min)
        return max_sum, min_sum
    # Step 3: Calculate the maximum cost for each node
    max_cost = float('-inf')
    for root in range(n):
        max_sum, min_sum = dfs(root, -1)
        max_cost = max(max_cost, max_sum - min_sum)
  return max_cos
```


C:\Users\RAJIN\Desktop\DAA\capstone.cpp - [Executing] - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Release
(globals)
Project Classes Debug
capstone.cpp
C:\Users\RAJIN\Desktop\DAA\capstone.exe
Compiler Resources
Abort Compilation
Shorten compiler paths
Line: 83 Col: 1 Sel: 0 Lines: 83 Length: 2214 Insert Done parsing in 0.656 seconds
31°C Haze
Search
ENG IN
19:19 09-09-2024

```
Enter the number of nodes: 4
Enter the prices for each node: 2
3
4
5
Enter the edges:
4
1
2
3
4
5
The maximum possible cost is: 5

--------------------------------
Process exited after 17.23 seconds with return value 0
Press any key to continue . . .
```

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\RAJIN\Desktop\DAA\capstone.exe
- Output Size: 129.9228515625 KiB
- Compilation Time: 2.22s
```

COMPLEXITY ANALYSIS

Best Case:

In the best case, the tree is balanced, such as a complete binary tree. Each DFS traversal explores roughly half the nodes at each level, efficiently balancing the workload across the tree. However, since DFS is still performed from each node, the overall time complexity remains O(n^2). The recursion stack depth is limited by the height of the tree, which is O(log n) for balanced trees, though this does not impact the total complexity significantly. Thus, the best-case time complexity is still O(n^2), and space complexity is O(n).

Worst Case:

In the worst case, the tree is skewed and resembles a linked list, where each node has only one child. Here, each DFS traversal explores the entire tree for each potential root, leading to O(n) time per traversal. Since DFS is performed for all `n` nodes, the total time complexity becomes O(n^2). The recursion depth also reaches its maximum of O(n) due to the skewed nature of the tree. The space complexity remains O(n) as it depends on the depth of the recursion stack and adjacency list.

Average Case:

In the average case, the tree is moderately balanced, with nodes distributed across various depths. Each DFS traversal still requires O(n) time because all nodes are explored in each traversal. Although the tree height is less than in the worst case, DFS is repeated from every node, keeping the total time complexity at O(n^2). The recursion depth will be somewhere between O(log n) and O(n), but it doesn't significantly affect the overall performance. Thus, both time and space complexities remain O(n^2) and O(n), respectively.

FUTURE SCOPE

Future work on this problem could focus on several key areas. Optimizing the algorithm for larger trees remains crucial, potentially through advanced data structures or parallel processing to improve efficiency. Another avenue is handling dynamic tree updates, enabling efficient recalculations when the tree structure changes. Extending the problem to weighted graphs with cycles or directed edges would broaden its applicability and present new challenges. Additionally, integrating the algorithm with real-world applications, such as network optimization and hierarchical resource management, could enhance its practical relevance. Exploring heuristic or approximation algorithms may also offer faster solutions for specific tree types or constraints, further improving performance.

CONCLUSION

In conclusion, solving the problem of maximizing the difference between the maximum and minimum price sums in an undirected tree involves significant computational challenges, particularly due to the need to evaluate all possible root nodes. The algorithm, with its O(n^2) time complexity, provides a thorough approach by performing depth-first search (DFS) from each node, ensuring accurate results. Despite its effectiveness, the algorithm's performance may be limited for very large trees, highlighting the need for optimization and potential advancements. Future research could focus on improving efficiency, adapting to dynamic tree structures, and extending the problem to more complex graph types. Overall, the problem offers valuable insights into tree-based path analysis with practical applications in various fields.