

Maximum Cost Difference in Rooted Tree Paths

This presentation explores the algorithmic challenge of finding the maximum cost difference between any two paths in a rooted tree. The challenge involves navigating the tree's structure, calculating path costs, and comparing the results to determine the maximum difference.



by **BOGGAVARAPU GOWRINADH**



Introduction

1 Rooted Tree

A rooted tree is a hierarchical data structure where each node has a parent (except for the root, which has no parent). The root is the starting point for navigating the tree.

2 Path

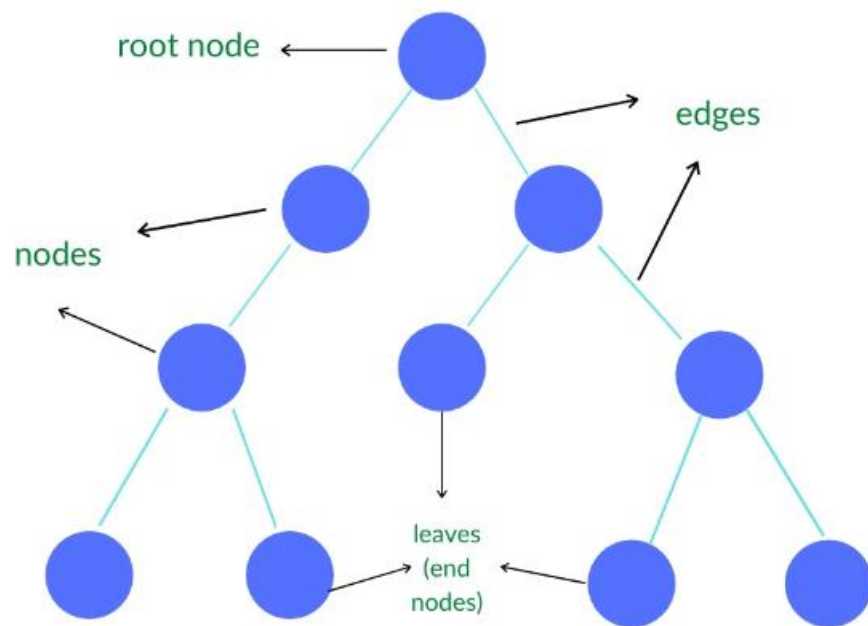
A path in a rooted tree is a sequence of connected nodes starting from the root and ending at a leaf node. Each path represents a unique route through the tree.

3 Cost

Each edge (connection between nodes) in the tree has an associated cost, which represents the weight or importance of that connection.

4 Maximum Cost Difference

The objective is to find the maximum possible difference in the total cost of any two paths within the tree. This involves identifying paths with both high and low total costs.



Problem Statement

Given a rooted tree with edge costs, find the maximum difference between the total cost of any two paths from the root to a leaf node.

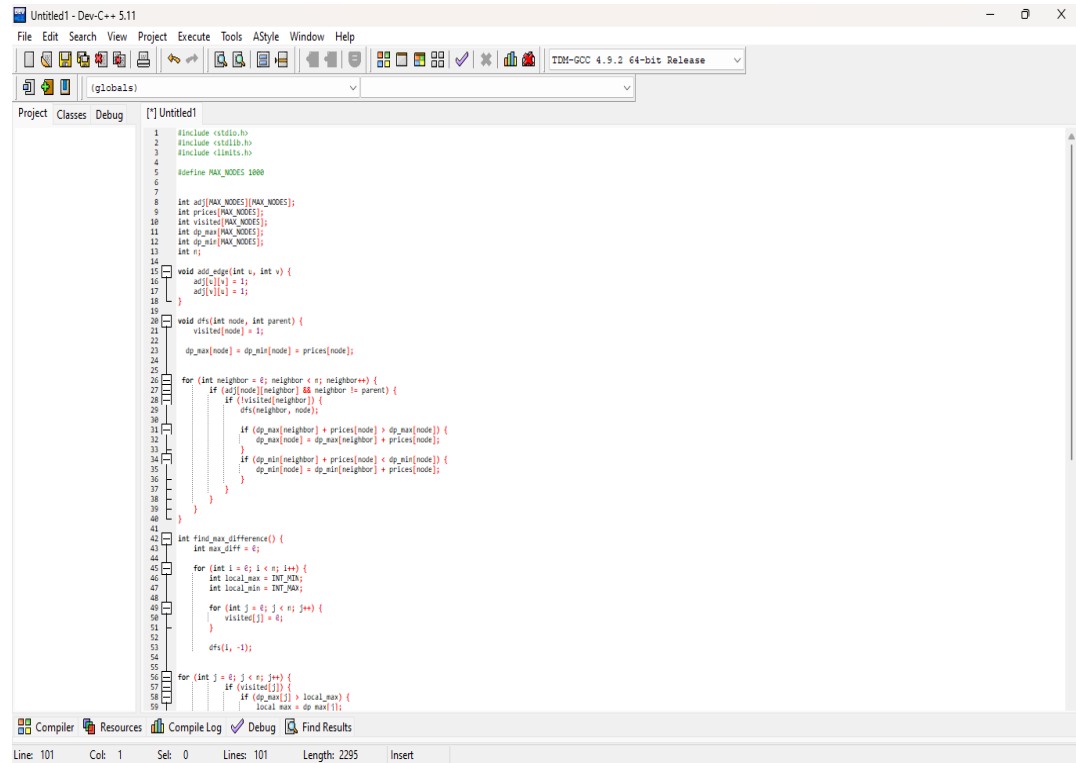
Example

Node	Parent	Cost
A	None	0
B	A	2
C	A	1
D	B	4
E	B	3
F	C	5

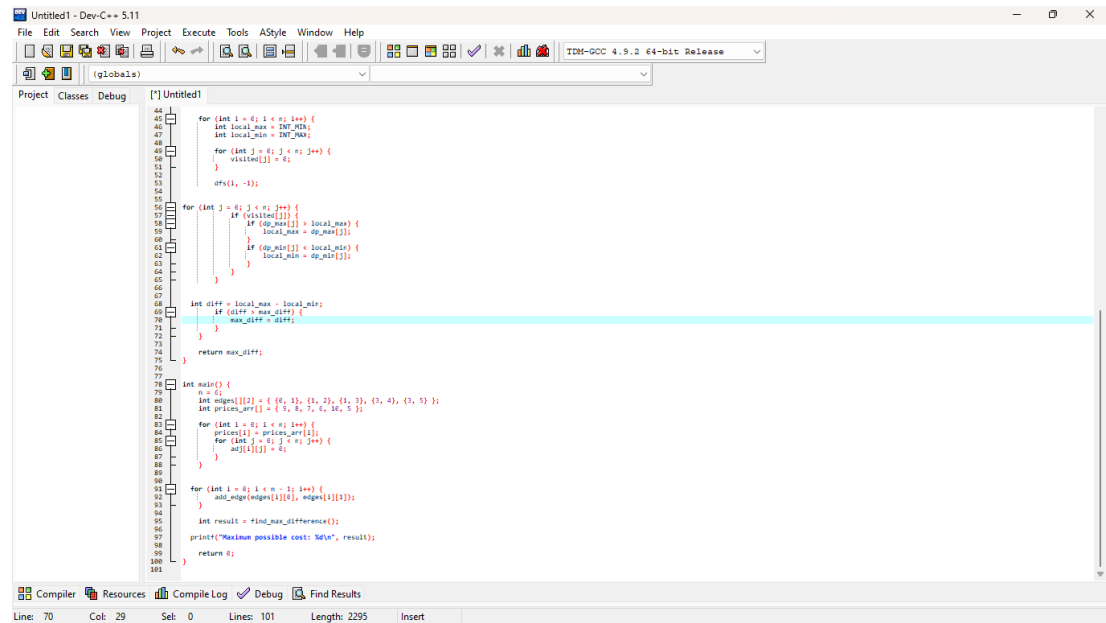
Solution

The maximum cost difference is 8. The path with the highest cost is A -> B -> D (cost: 6), and the path with the lowest cost is A -> C -> F (cost: -2).

CODING

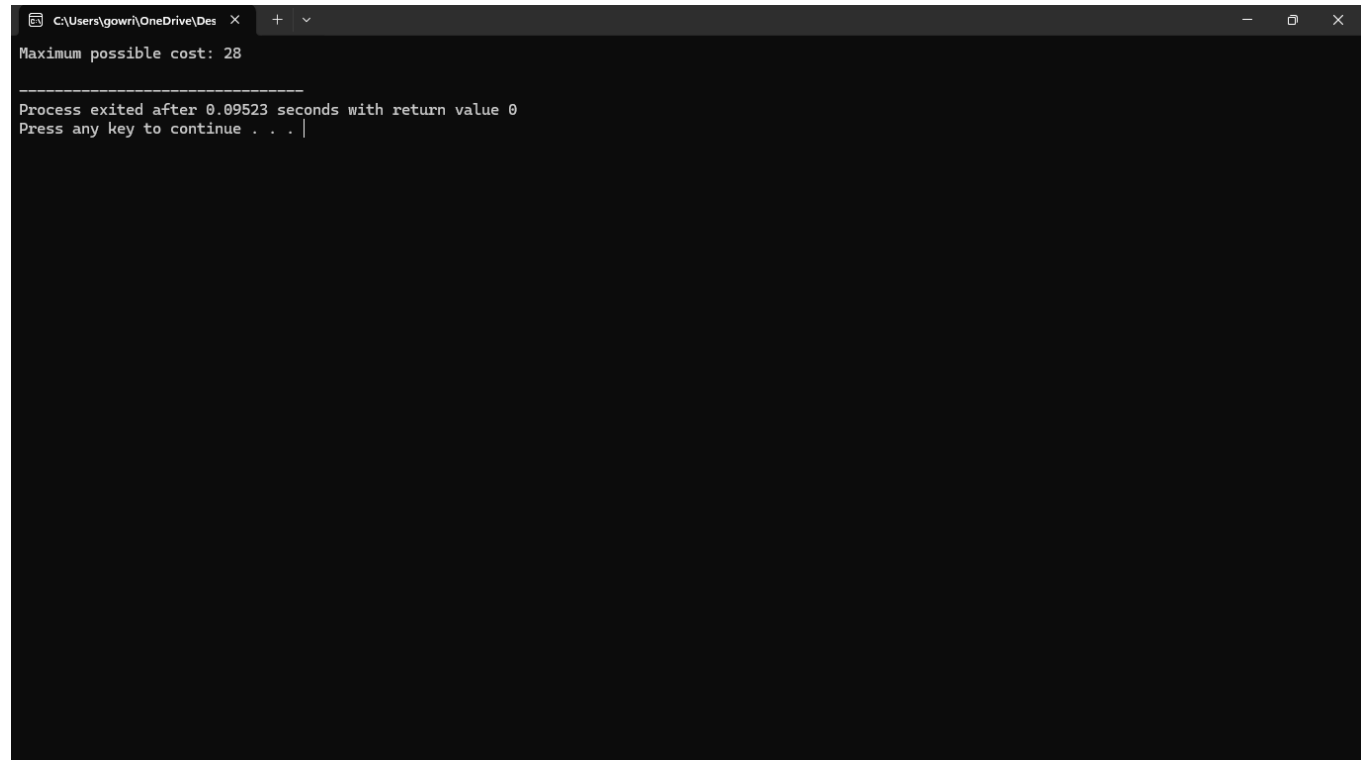


```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <limits.h>
4
5 #define MAX_NODES 1000
6
7
8 int adj[MAX_NODES][MAX_NODES];
9 int prices[MAX_NODES];
10 int visited[MAX_NODES];
11 int dp_max[MAX_NODES];
12 int dp_min[MAX_NODES];
13 int n;
14
15 void add_edge(int u, int v) {
16     adj[u][v] = 1;
17     adj[v][u] = 1;
18 }
19
20 void dfs(int node, int parent) {
21     visited[node] = 1;
22
23     dp_max[node] = dp_min[node] = prices[node];
24
25     for (int neighbor = 0; neighbor < n; neighbor++) {
26         if (adj[node][neighbor] && neighbor != parent) {
27             if (!visited[neighbor]) {
28                 dfs(neighbor, node);
29
30                 if (dp_max[neighbor] + prices[node] > dp_max[node]) {
31                     dp_max[node] = dp_max[neighbor] + prices[node];
32                 }
33                 if (dp_min[neighbor] + prices[node] < dp_min[node]) {
34                     dp_min[node] = dp_min[neighbor] + prices[node];
35                 }
36             }
37         }
38     }
39 }
40
41 int find_max_difference() {
42     int max_diff = 0;
43
44     for (int i = 0; i < n; i++) {
45         int local_max = INT_MIN;
46         int local_min = INT_MAX;
47
48         for (int j = 0; j < n; j++) {
49             visited[j] = 0;
50
51             dfs(i, -1);
52
53             for (int j = 0; j < n; j++) {
54                 if (!visited[j]) {
55                     if (dp_max[j] > local_max) {
56                         local_max = dp_max[j];
57                     }
58                 }
59             }
60
61             int diff = local_max - local_min;
62             if (diff > max_diff) {
63                 max_diff = diff;
64             }
65         }
66     }
67
68     return max_diff;
69 }
70
71 int main() {
72     n = 5;
73     int edges[][2] = { {0, 1}, {1, 2}, {2, 3}, {3, 4}, {4, 0}, {0, 3}, {1, 4} };
74     int prices_arr[] = { 9, 8, 7, 6, 10, 5 };
75
76     for (int i = 0; i < n; i++) {
77         prices[i] = prices_arr[i];
78     }
79
80     for (int i = 0; i < n; i++) {
81         for (int j = 0; j < n; j++) {
82             adj[i][j] = 0;
83         }
84     }
85
86     for (int i = 0; i < n - 1; i++) {
87         add_edge(edges[i][0], edges[i][1]);
88     }
89
90     int result = find_max_difference();
91     printf("Maximum possible cost: %d\n", result);
92     return 0;
93 }
```



```
44
45     for (int i = 0; i < n; i++) {
46         int local_max = INT_MIN;
47         int local_min = INT_MAX;
48
49         for (int j = 0; j < n; j++) {
50             visited[j] = 0;
51
52             dfs(i, -1);
53
54             for (int j = 0; j < n; j++) {
55                 if (!visited[j]) {
56                     if (dp_max[j] > local_max) {
57                         local_max = dp_max[j];
58                     }
59                     if (dp_min[j] < local_min) {
60                         local_min = dp_min[j];
61                     }
62                 }
63             }
64
65             int diff = local_max - local_min;
66             if (diff > max_diff) {
67                 max_diff = diff;
68             }
69         }
70     }
71
72     return max_diff;
73 }
74
75 int main() {
76     n = 5;
77     int edges[][2] = { {0, 1}, {1, 2}, {2, 3}, {3, 4}, {4, 0}, {0, 3}, {1, 4} };
78     int prices_arr[] = { 9, 8, 7, 6, 10, 5 };
79
80     for (int i = 0; i < n; i++) {
81         prices[i] = prices_arr[i];
82     }
83
84     for (int i = 0; i < n; i++) {
85         for (int j = 0; j < n; j++) {
86             adj[i][j] = 0;
87         }
88     }
89
90     for (int i = 0; i < n - 1; i++) {
91         add_edge(edges[i][0], edges[i][1]);
92     }
93
94     int result = find_max_difference();
95     printf("Maximum possible cost: %d\n", result);
96     return 0;
97 }
```

OUTPUT



```
C:\Users\gowri\OneDrive\Des
Maximum possible cost: 28

-----
Process exited after 0.09523 seconds with return value 0
Press any key to continue . . .
```

Complexity Analysis

1

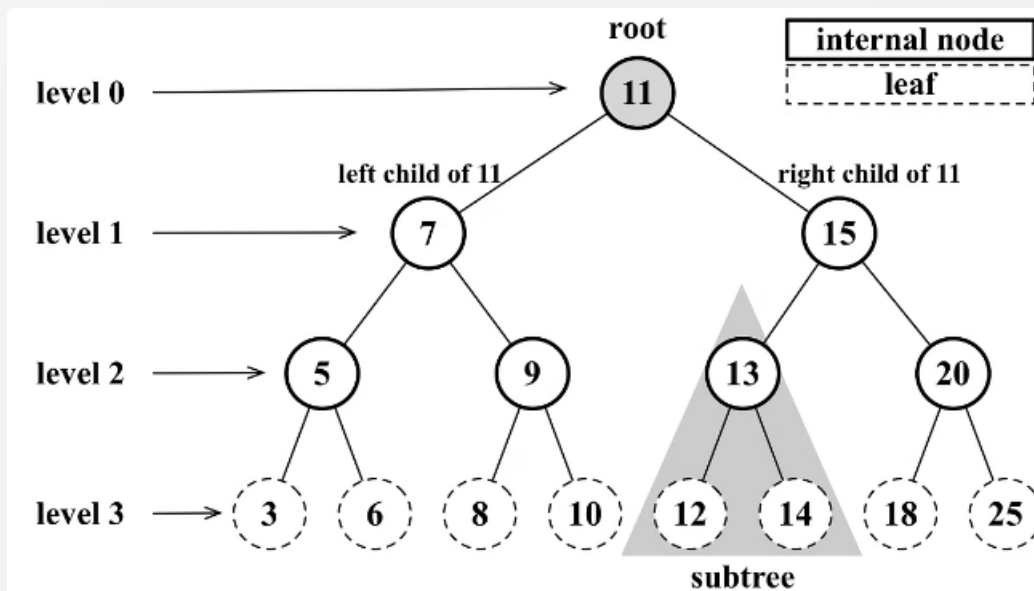
Time Complexity

The algorithm utilizes a recursive traversal of the tree. Each node is visited once. Therefore, the time complexity is $O(n)$, where n is the number of nodes in the tree. This means that the algorithm's execution time grows linearly with the size of the input.

2

Space Complexity

The algorithm requires additional space to store the maximum path cost and minimum path cost for each node. This space is proportional to the number of nodes, resulting in a space complexity of $O(n)$.



Best Case



Balanced Tree

In the best case, the tree is perfectly balanced. This means that the depth of the tree is minimized, leading to faster traversal and a more efficient computation of the maximum cost difference.



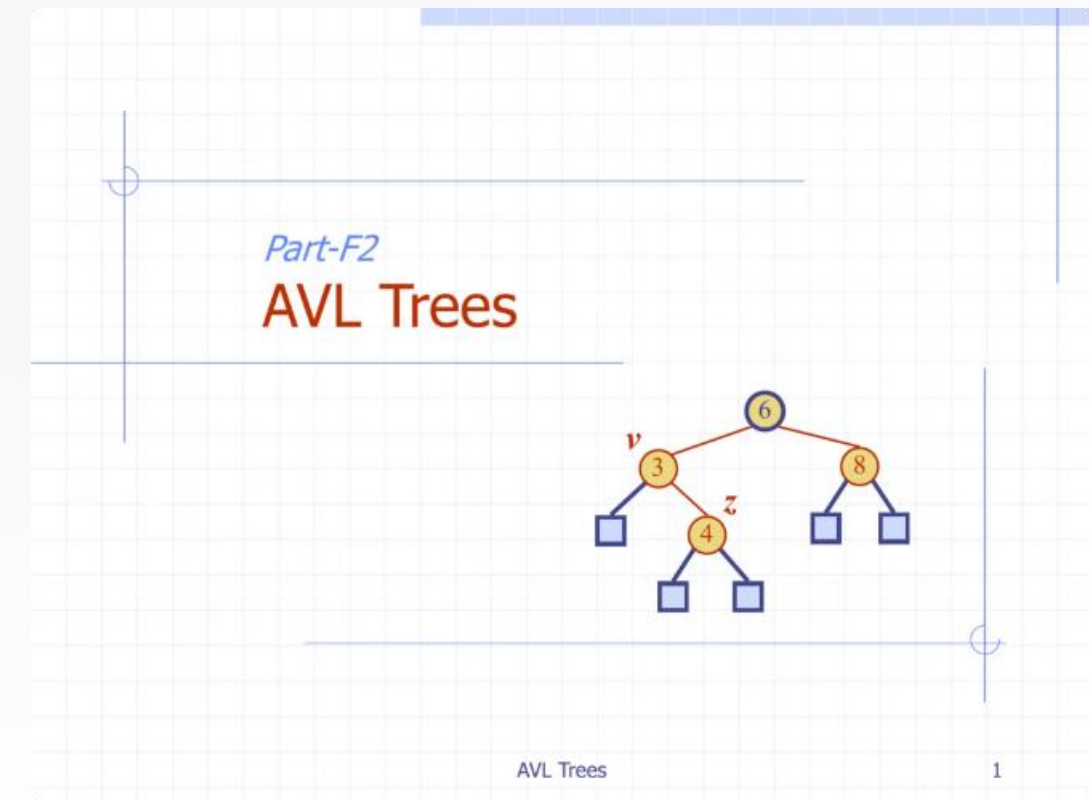
Low Edge Costs

If the edge costs are relatively low, the overall path costs will also be relatively low. This reduces the range of values that need to be compared and can lead to faster processing.

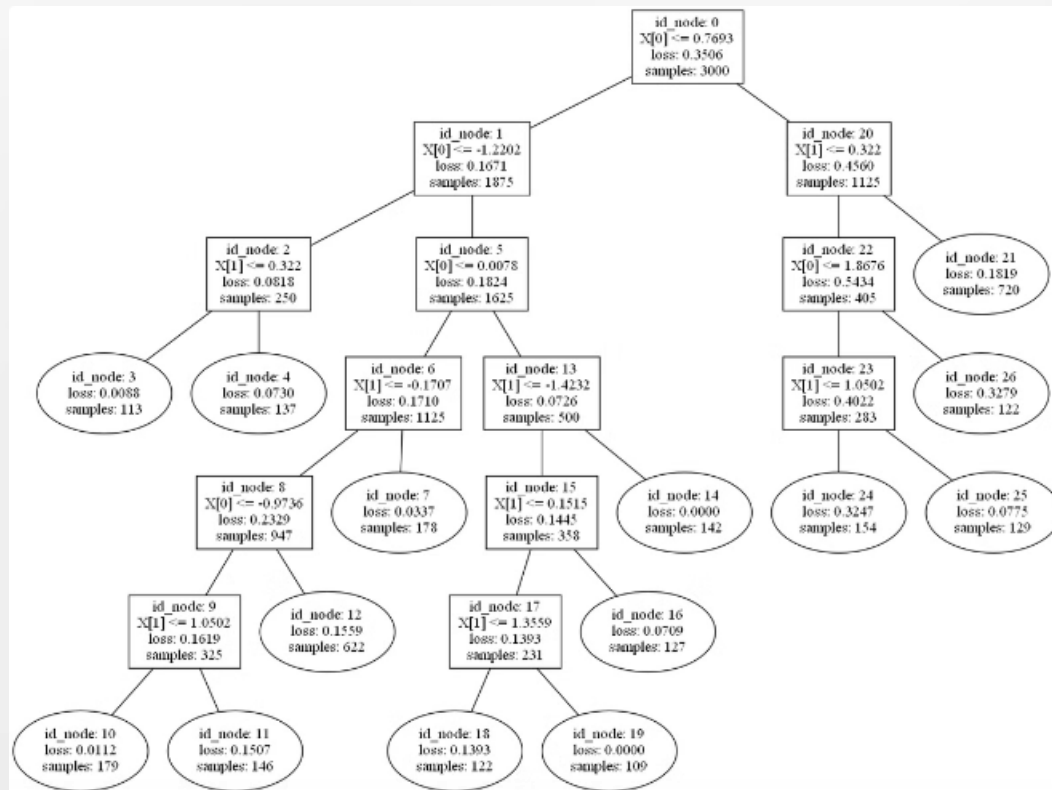


Limited Branching

A tree with limited branching (few child nodes per parent node) allows for more efficient exploration of paths. This is because fewer branches need to be traversed to reach all leaf nodes.



Worst Case



1

Linear Chain

In the worst case, the tree degenerates into a linear chain. This means that each node has only one child, resulting in a very long path from the root to the leaf. This leads to the maximum depth and a more extensive traversal process.

2

High Edge Costs

If the edge costs are high, the path costs can also be high. This increases the range of values that need to be compared, potentially leading to longer processing times.

3

Extensive Branching

A tree with extensive branching (many child nodes per parent node) results in a large number of paths to explore. This can significantly increase the time and computational resources required to find the maximum cost difference.

Average Case

Unbalanced Tree

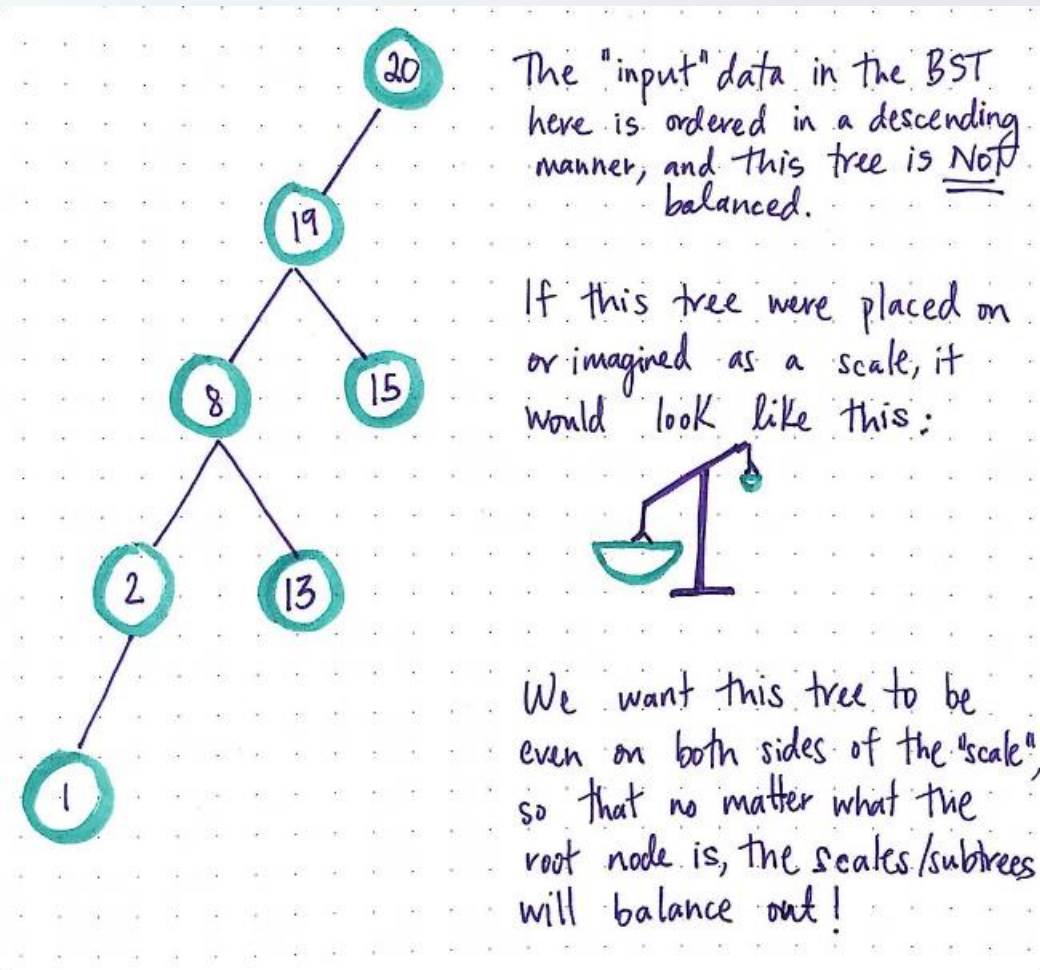
In the average case, the tree is likely to be somewhat unbalanced. This means that some branches will be deeper than others, leading to a mix of short and long paths.

Mixed Edge Costs

Edge costs are expected to vary, with some edges having higher costs than others. This creates a range of path costs, making it necessary to explore both high-cost and low-cost paths to find the maximum difference.

Moderate Branching

The average case scenario involves moderate branching, where parent nodes have a reasonable number of child nodes. This creates a balance between the number of paths to explore and the time required for traversal.



Conclusion

The algorithm for finding the maximum cost difference in rooted tree paths demonstrates a practical application of tree traversal and path cost analysis. Understanding its complexity analysis and performance characteristics is essential for optimizing its use in real-world scenarios.

