Rahul Thapa

I used a modified version of Dijkstra algorithm. First, I created a helper array which contains all the nodes from the triangle, traversed top to bottom, left to right. Each of those nodes has additional information: the value of the node, its index in original triangle, path sum, and parent vertex. Unlike usual Dijkstra algorithm, which starts from a starting node value 0, I start my algorithm with the path value equal to the value of the node. After that, it is the standard Dijkstra algorithm, where I check two of its children, update the path sum of those children, then check the path sum value of the entire nodes in my helper array, pick the lowest path sum node, and repeat the process. I do this until I reach at the end of the triangle and find that the path sum at that point is less than any other path sum.

Here are some of the test cases that I used in my experiment. In the tuple, the first value is the triangle, second value is the minimum path value, and third value is the system time it took to run.

1. (1, [[1]], 6.4373016357421875e-06)
2. (3, [[2], [2, 1]], 1.0967254638671875e-05)
3. (17, [[7], [9, 5], [6, 5, 7]], 1.8596649169921875e-05)

(120,

[[27],

[45, 9],

[16, 34, 3],

[26, 32, 26, 63],

[33, 4, 27, 12, 3],

[21, 15, 22, 51, 55, 39],

[15, 31, 36, 3, 16, 15, 3],

[24, 63, 37, 55, 3, 47, 16, 47]],

7.867813110351562e-05)

All these test cases pass.

1. The algorithm is given in the attached python file.
2. The algorithm’s order of efficiency is O(). This is because in my algorithm, I have an outside while loop, which in worse case goes through every node, which takes (This is the worst case. Normally, It will be less than that.). For each one of those nodes, I am going through the entire array that I use to maintain every node’s current state. So that is additional for each iteration. Therefore, the total time complexity is O().

Note: I am essentially using a modified version of Dijkstra algorithm where the graph is represented as weight matrix and priority queue implemented as sorted array. In such as case, the time complexity of such algorithm is O(), where V is the number of nodes. However, in our case, V = . Therefore, the total time complexity of the algorithm is O().

1. The scatter plot is given below. We can see that the plot is growing more than linearly but certainly less than exponentially. I hypothesize it to be some polynomial degree.

Looking closely at the values, I found out that the curve is increasing with the polynomial degree of 4. I figured that out by taking two points and checking time complexity relation, which holds true.

(I know that you mentioned it is better to choose geometric progression rather than arithmetic. However, it was not giving me smooth curve with less value of n and for large values, it was taking long time. Therefore, to make sure we can see the trend, I chose this.)

A screenshot of a cell phone

Description automatically generated

1. For my system, the largest value of n for which the algorithm can solve the problem in 1 minute is approximately 149. Note: This is the value based on my algorithm which has the time efficiency of Θ().
2. Note: The calculation below is based on my algorithm and my system specification. From my plot, I found out that for my system, it takes approximately 12.164 seconds to solve the problem with base n=100. We know that the order of growth of the algorithm is given as Θ(). Therefore,

When t = 24 hours = 86400 secs

Solving for n, we get approximately 918.

When t2 = one year = 31556952 secs

Solving for n, we get approximately 4013.