Lumberjack Problem

DSA Project Report

Group 14
Bhargav (200010010)
Abhishek (200010021)
M V Karthik (200010030)
Om Patil (200010036)

November 2021

1 Algorithm

1.1 Description

1.1.1 Overview

Our algorithm is essentially a greedy approach to the problem. We navigate to the tree which provides us with the highest rate and cut it in the optimal direction. We define rate as the amount of value/price we obtain by cutting that tree (including any value obtained due to domino effect) divided by time required to travel to and cut the tree. The optimal direction is defined as the direction which causes maximum value to be obtained by domino effect. We repeat this procedure until we have time left

1.1.2 Flow

At the start of the program we first input all the data which is provided for the sample case. Post this we calculate the rate for all the trees (which are not cut) based on our current position. We then navigate to the tree which provides the highest rate, can be travelled to and cut within the remaining time. Once we arrive at the tree we cut the tree in the optimum direction. We then repeat this process until no tree remains which can be cut in whatever time is left.

1.1.3 Calculating Affects

Affects are a number defined for each tree for each direction which indicates the value/profit which can be obtained due to domino effect of the tree being cut in that direction.

To calculate affects we start from the very next point to the tree in the required direction, post which we start checking each space one by one (till the height-1 spot after the tree) till we find an uncut tree. If the tree is of less weight then we add its value to the affects, we then also add the affects of this tree in the same direction to the affects of the original tree and then stop. Otherwise if the tree has less weight than the original tree then we stop right there without adding anything.

To optimize the algorithm we calculate these affects in a interesting fashion. Whenever we require an affects for a specific tree and direction we check whether the affects has been calculated already for that specific case if yes we return it if no we calculate it in the method mentioned above and save it for the future. This is achieved through the return affects function. As the affects are dependant on domino effect and in turn trees we empty the stored effects every time a tree is cut as it is now obsolete for the current situation.

1.1.4 Calculating Rate

When calculating the rate for a tree the first thing we do is update the time property of the tree, which stores the time required to navigate to and cut the tree, to whatever the value is based on the current position. Then we calculate the rate by adding the value of the tree to the highest affects of all the directions and divide by the time calculated in the previous step. This rate is then stored in the rate property of the tree.

1.1.5 Finding and Navigating to the Highest Rate Tree

To find the highest rate tree we sort trees based on the rate using numpy (quicksort). We then consider the top tree one by one until we find a tree which has not been cut and can be cut in the remaining time. If we find such a tree we go ahead and navigate to it by moving such that we match the x value of the target tree and then we match the y value of the target tree. In case we do not find any tree which satisfies our conditions we end the program as either all trees have been cut or no tree can be cut in the remaining time.

1.2 Pseudocode

```
function $returnAffects(tree, dir)$
    if !$tree.affects[dir]$ then
        for $i = 1$ to $tree.height$ in $dir$ do
            if $newTree$ at $tree.pos + i$ then
                if $newTree.weight < tree.weight$ then
                    add $newTree.value$ to $tree.affects[dir]$
                    add $returnAffects(newTree)$ to $tree.affects[dir]$
                    break
                else
                    break
                end if
            end if
        end for
    end if
end function

function $dominoValue(tree)$
    find $max(returnAffects(tree, dir))$
    $tree.optDir \leftarrow dir$
    return $returnAffects(tree, dir)$
end function

function $greedyEvaluateAll$
    for all $tree$ do
        if !$tree.cut$ then
            $tree.time \leftarrow abs(pos.x - tree.x) + abs(pos.y - tree.y) + tree.thickness$
            $tree.rate \leftarrow (tree.value + dominoValue(tree))/tree.time$
        end if
    end for
end function

function $greedyNavigate$
    $QuickSort(trees, tree.rate)$
    for all $tree$ in $trees$ do
        if !$tree.cut$ and $timeLeft(tree.time)$ then
            $target \leftarrow tree$
            break
        end if
    end for
    if !$target$ then
        exit
    end if
    $move$ to $(target.x, target.y)$
end function
```

```
function $greedyCut$
    $tree \leftarrow tree$ at $pos$
    $cut$ in $tree.optDir$
end function

while $isTimeLeft$ do
    $greedyEvaluateAll$
    $greedyNavigate$
    if $isTimeLeft$ then
        $greedyCut$
        for all $tree$ do
            $tree.affects \leftarrow 0$
        end for
    end if
end while
```

1.3 Data Structures

1.3.1 Tree

This is the class which stores all the information of a tree. Given information is populated here on initializing object. To reduce the time complexity tree value and tree weight are calculated and stored at time of instantiating object. Rate and time are also stored here post initializing.

1.3.2 Forest

This is an array of objects trees which have same order as input. It helps in accessing a tree from its index.

1.3.3 Map

Map is 2D numpy integer array. Value at (x,y) indices of map contains the index of tree in Forest array if tree is present at (x,y). If there is no tree at a position then the value will be -1. This will help to check if there is a tree at a given position and find its index if it is in $O(1)$ time complexity.

1.3.4 pos_x,pos_y

pos_x and pos _y are global variables which contain the current x and y coordinates of lumberjack at any given time.

1.4 Complexity

Memory complexity of overall program : $O(N^2 + K)$
Time complexity of overall program : $O(K^3 + KN)$

Each of the below functions are called in series each cycle and we can have max k cycles.

1.4.1 greedyEvaluateAll

Time complexity : $O(K)$

1.4.2 greedyNavigate

Time complexity : $O(K^2 + N)$

1.4.3 greedyCut

Time complexity : $O(N)$

2 Observations

- Cut tree has to fall only inside the grid.
- Domino effect is only valid when the distance between two tree is at maximum one unit less than height of falling tree.
- Weight of falling tree has to be strictly greater than tree on which it falls to cause domino effect.
- Appending trees to affects list has more time and memory complexity than adding its value to a affects variable.

3 Tried Approaches

3.1 Simulation Greedy

Calculating and storing a list of all trees which can be cut in given time(Simulation). After calculating the list of trees then we start navigating to trees in list greedily such that we can save time.

But we underestimated the effect of order of cutting trees where domino effect relies on. So we ended up getting low profits than non-simulated greedy one.

3.2 Multi-level Domino

Currently, while calculating the rate of tree we only consider one-level of domino effect i.e if a tree is falling we only consider first-nearest tree's value which is falling. We ignore any other trees which are after first tree. We tried considering rest of trees too. But this resulted in more time complexity(probably due to recursive dominoValue function) and less profits.

Our whole source tree of codes we submitted is hosted on github.

4 Possible Methods

One possible method to try is to store all trees which we can cut through greedy approach and store it in a list. Then start navigating and cutting the trees which stored in such a way that we don't alter the domino effects.

Another abstract method is one which works similar to a gravity field. The lumberjack should move in the direction in which the projection from net vector of gravity forces ($value/distance^2$) from each tree is highest. This would ideally lead him to the most value dense area of the forest hopefully giving him the most profit in least time.

One method which fails in time complexity is a brand and bound approach which tries all possible permutations to find the permutation which provide most profit in given time. This approach would guarantee maximum profit but would probably exceed time limits.