Directing Rainfall

Pascal Engel
University of Tuebingen
pascal.engel@student.uni-tuebingen.de

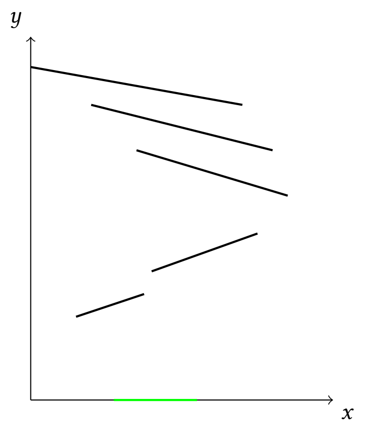

Figure 1: Sample input

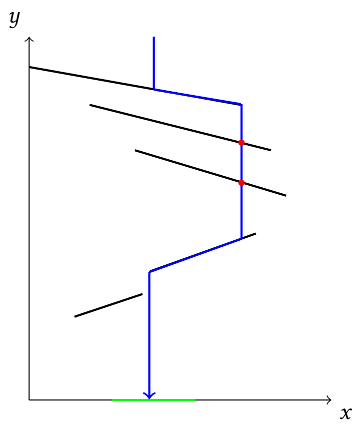

Figure 2: An optimal solution

ACM Reference format:
Pascal Engel. 2016. Directing Rainfall. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 4 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In this text I will present and discuss a possible solution for the problem F of the 2019 ACM International Collegiate Programming Contest "Directing Rainfall" using a custom pseduo-sorting algorithm and rather simple list recursion in Haskell.

2 PROBLEM DEFINITION

The exercise is set in the wine region of the Douro valley, which is frequently visited by tourists. In order to protect tourists from the vast amount of sunshine in broad daylight, sun tarps are installed above the vineyard. Since those sun tarps won't allow rain to reach the vineyard though and could endager the vine harvest, our task is to find the minimum amount of punctures to be put into them so that rain from above can reach the ground.
In order to simplify the problem, some constraints are given: The vineyard and the tarps lay in a two-dimensional plane [fig. 1]. The vineyard is represented by a range on the x-axis and tarps are line segments between two points above the x-axis. Rain originates only above the vineyard and falls vertically (parallel to y-axis) from infinitely high. Once it reaches a tarp, rain flows down towards its lower end [fig. 2]. We may assume that a lower point is always defined because tarps may not be horizontal. Collisions should also not be permitted, so we won't encounter any intersections of tarps. Furthermore all coordinates are given as discrete integer values.

Conference'17, Washington, DC, USA
2016. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

3 OUTLINE

To solve the problem, first we need to parse the input. Then, for computation, it seems intuitive to find some sort of topological order in which rain may reach the tarps. This can then be used to simulate or predict the directed rainfall in various ways depending on whether punctures are being put. Finally it will turn out to be useful to visualise the input and parts of the computation.
If such an order can be found, lists are a useful data structure to store the result and can be processed recursively to compute the desired optimal solution. Since an optimal solution is the minimum number of punctures to reach the vineyard, partial solutions, that can be used recursively, are the minimum numbers of punctures to reach tarps. Considering this, Haskell is a fitting tool to fulfil the task because as a functional programming language it is great in handling both lists and recursion. Additionally as will be shown, the type class system, particularly the `Maybe` monad, provides us with a useful structure to model special behaviour.

4 PARSING

The following Haskell data types describe the programs input. The parser is generated using the Happy system. Happy is given a simple context-free grammar in BNF describing the input and producing a result in a Haskell data type as such:

```
data Input = Input
    { vineyard  :: Range
    , noOfTarps :: Int
    , tarps     :: [Tarp]
    }
data Tarp = T Point Point
```

Types `Range` and `Point` are both simple tuples of integers. The field `noOfTarps` can actually be ignored because `length tarps ==`

`noOfTarps` holds for any valid input. It is probably only required in the exercise to make it easier to initialise an array, which we don't use here anyways.

5 SORTING

Our goal is to sort the input tarps in a way so that for any tarp t in the resulting list it holds that no tarp that overlaps t is placed after t in the list. This roughly corresponds to a topological order in graph theory. To achieve this, at least a criterion for comparison of two tarps of type `Tarp -> Tarp -> Bool` to determine which one will be reached first is required. If we can define this, we can also define an instance of `Ord Tarp`, which would help us (using the module `Data.List`) to sort the whole list of tarps.
To define which one of two tarps t_1 and t_2 is 'upper' we need to consider three (non-exclusive) cases in this order:

(1) There is no overlap (on the x-axis) of the two tarps.
(2) The lower point of tarp t_1 is greater or equal on the y-axis than the upper point of tarp t_2.
(3) Any other case, where an overlap exists. (e.g. in fig. 3)

(1) cannot be defined without some ambiguity. I will come back to this after discussing the other cases. (2) yields an obvious solution: If there is an overlap and every point of t_2 is higher than every point of t_1, than t_2 must be above t_1. Comparing the tarps in case (3) gets a little more complicated because only some points of the upper tarp t_2 are above some points of t_1. Since all values are discrete integers, we could simply compare all of them in pairs. But this is very inefficient because coordinates of points which are not endpoints of tarps need to be computed first. Instead, depending on the type of overlap that two tarps create, there is always one point that needs to be above the geometric line that is going through the points of the lower tarp. In the example in fig. 3 the lower point of t_2 has to be above the line going through t_1 in order for t_2 to be higher. So we calculate the line $y = m * x + b$ for t_1, replace x with the x-value of the lower point of t_2 and compare the result with the actual y-value of the lower point of t_2. We don't have to consider t_2's upper point because, if it were below the line, t_2 would intersect t_1, violating our constraints. Using this method, we can define the upper-relation for all tarps that overlap.
In (1) where there is no overlap we can only arbitrarily define the relation, e.g. the tarp with the bigger x-coordinate is 'upper'. This however poses a problem, when there are finally more than two tarps. For example, if we arbitrarily define $t_1 > t_2$ (because t_1 and t_2 don't overlap) but there is another tarp t_3 with $t_3 > t_1$ and $t_2 > t_3$, then the relation is circular, eventually preventing the computation to terminate. So arbitrarily defining one tarp as 'upper', when tarps don't overlap cannot be the solution for this. Instead the sorting algorithm should ignore this case. Intuitively tarps cannot be compared in terms of one being reachable from the other when there is no reachabilty at all, due to a lack of overlaps. We can achieve this by altering the type of the comparison criterion to `Tarp -> Tarp -> Maybe Bool`. Following this, the result of comparing two tarps that don't overlap would be `Nothing`, the result for overlapping tarps would simply be wrapped in a `Just`.

Using this strategy imposes another problem though: The criterion doesn't satisfy transitivity as demonstrated in fig. 4. In this example,

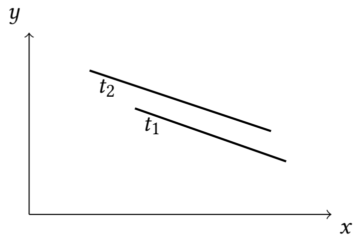

Figure 3: Overlapping tarps

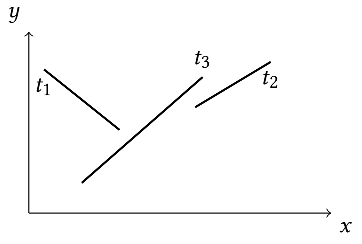

Figure 4: Transitivity is not implied

we can deduce $t_1 > t_3$ and $t_3 > t_2$ but not $t_1 > t_2$. However, transitivity is required for efficient sorting algorithms like quicksort or mergesort. Strictly speaking, this even prevents us from defining a total order but that is not what we are looking for to begin with. As stated above, the 'sorted' list should only hold the condition "that for any tarp t in the resulting list it holds that no tarp that overlaps t is placed after t in the list".
To implement this idea, we use an algorithm very similar to max-sort, but instead of selecting only one maximum element and inserting it into the sorted list, we select all tarps that are not overlapping with any other or are above any other. So in each recursion step we select at least one up to all tarps as `max`, which can improve the runtime significantly, however in worst case it is still in $O(n^2)$ like max-sort. The implementation in Haskell pseudocode looks like this:

```
sort :: [Tarp] -> [Tarp]
sort [] = []
sort ts = sort (ts\\max) ++ max
 where
  max = [t | t<-ts, all (fromMaybe True . t >) $ ts\\[t]]
```

This finally computes the desired pseudo-order in the list of tarps, which satisfies the specified condition. With this part being done, we can proceed to *simulate* the flow of rainwater down the tarps.

6 CALCULATING COSTS

For each tarp in the sorted list, we want to assign some cost c representing the minimum number of punctures required to reach it, which eventually will lead us to the solution of minimum punctures required to reach the vineyard. Some tarps may be unreachable, meaning c can also be `Nothing`, so costs are of type `Maybe Int`.
Before we can do that however, we need to consider tarps that are only partially reachable (e.g. the left side of the upmost tarp in fig. 7

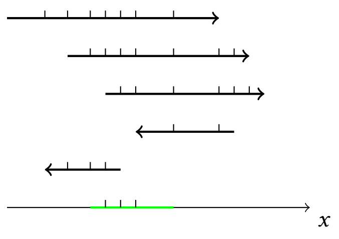

Figure 5: Ordered, simplified and split tarps

can never be reached by water because water only flows to the right). In this case we need to split them into ranges representing reachable and unreachable parts. Furthermore, different ranges of the same tarp may have different costs assigned. The simplest way to tackle this, is splitting all tarps based on all tarps' end points and assigning the resulting list of ranges to each tarp. For this, we also include the vineyard as an imaginative tarp. This makes the following computation easier because now ranges always lay directly above each other and don't overlap partially.
Also, because the tarps are already in a desired order, their y-coordinates become irrelevant, so we may just forget about them. Instead, we assign to each an orientation either to the left or to the right (indicated by an arrowhead). The program also provides us with such a visualisation as seen in fig. 5.

The resulting data finally is allowing us to to calculate costs for each range. The cost c of a range $[a, b]$ on tarp t depends on various factors: Without loss of generality we can assume that t is orientated towards the right. This means water on the tarp flows from left to right. As also shown in fig. 6, c in this case depends on ranges with their costs assigned to them respectively, which can be assumed as already computed because the computaion is recursive (if the relevant range is not present, the corresponding cost c_i will be `Nothing`):

(1) The cost c_1 of the range $[a', b']$ on the same tarp with $a = b'$.
(2) The cost c_2 of a range $[a', b']$ on the nearest above tarp with $a = b'$, where b' is the lower point of that tarp (meaning water drops off the edge, visually represented by an arrowhead).
(3) The cost c_3 of a range $[a', b']$ on the nearest above tarp covering t with $a = a'$ and $b = b'$

Considering these recursively already computed values, we can assign $c = min(c_1, c_2, c_3 + 1)$, where c_3 is incremented because from the corresponding range we have to put a puncture in order to reach $[a, b]$ on t. The *min* function has a special behaviour as it works with values of type `Maybe Int`. Unlike its applicative version, it always returns an `Int` (wrapped in a `Just`) unless *all* input values are `Nothing`.
Analogously, if tarp t is orientated towards the left, costs are calculated as stated above, with the relevant ranges turned around. Knowing this, we can recursively go through all tarps from top to bottom and all their ranges starting with the left-most if the tarp is orientated to the right and vice-versa. However, because all costs depend on previously calculated costs, we need to initialise

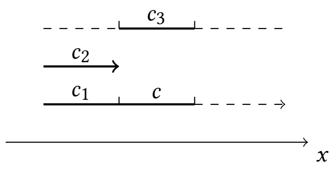

Figure 6: Recursive dependency of cost c

some costs before we can begin the recursion, else all costs would be `Nothing`. We initialise the computation by inserting another imaginative tarp at the top of the list representing rain coming from directly above the vineyard (so this tarp will have the same range as the vineyard). It is split into ranges the same way as the regular tarps and all ranges will be set to `Just 0`. This guarantees that the costs of all following tarps are computed as if rain is falling from infinitely high (as specified in the description).

Now we are finally able to compute the costs of all ranges of all tarps including the vineyard. For this, we add an additional tarp at the end of the list, representing the vineyard. The two imaginative tarps representing the rain from directly above and the vineyard itself are represented by a special sort of tarp. That is because unlike regular tarps they do not have a lower end where rain can flow towards to. This means water on those tarps will never flow from one range to a neighbouring range. This has to be considered in special cases when computing costs of ranges.
Finally, when all costs are computed, the solution to the problem is the minimum cost of the last tarp in the list (i.e. the vineyard). In the example in fig. 7 this value is 2 meaning we can reach that part of the vineyard with at least 2 punctures put in tarps above. With the knowledge of how to assign costs, we are also able to reconstruct all optimal paths that water will flow in a bottom-up way. Note that, unlike the possible solution presented in the beginning [fig. 2], the punctures do not need to be on a specific coordinate, but rather can be chosen more or less arbitrarily inside a range. That is another reason why the model of ranges is very useful here. For the sake of simplicity, we can assume that punctures are being put in the middle of a range, this ensures that water can drop from puncture to puncture. Reconstructing the optimal path is however not part of the program because the challenge was only to show the minimum number of punctures required.

7 VISUALISATION

The program has a command line interface and will return the solution as a simple integer, when given an input file that is structured by the specified rules. Additionally, with the module `Exports.hs` there are several options to visualise the input as well as the necessary steps towards it as explained so far. This is very helpful when debugging or in order to understand potentially unexpected results. With the flag `-i` the program will show the tarps unprocessed in a two-dimensional plane as shown in fig. 1. This is helpful when testing sample input files because the program does expect well-formed inputs and rather calculates flawed results than exiting with an error. Checking whether an input is ill-formed would unnecessarily increase the runtime and therefor is left out. Intersections and other violations of constraints need to be sorted out manually.

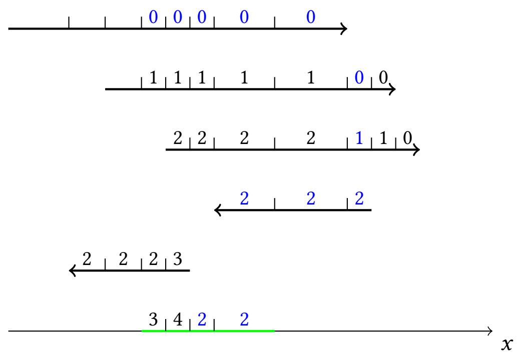

Figure 7: Final computed costs, blue labels represent ranges which are part of an optimal path

The flag -s will show the sorted and simplified list of tarps as demonstrated in fig. 5. This is especially helpful when checking the correctness of the calculated order.
Finally, with the -w flag the weighted tarps with all costs assigned are displayed. Unlike in fig. 7 the optimal paths are not being shown, however those can be comprehended manually or the functionality might be implemented as an additional option in further work.
The visualisation will be displayed in a browser using a html-file. The module `Exports.hs` wraps all relevant data in a record that can be easily transformed to a JSON-Object using the Aeson library for Haskell. The generated JSON-file is then embedded in a html-file with a blank canvas object, in which a script will draw and scale the exported data and present it in the format shown above.

8 DISCUSSION

It has been shown that the algorithm works, however there are others approaches the problem could have been solved, specifically using graphs. Since we are talking about reachability and costs of reaching a specific part of a tarp, it could be useful to structure tarps or ranges as vertices in a weighted directed acyclic graph. This could potentially lead to more efficient computation because only tarps that are reachable from another will be considered. However, it seems likely that computing such a graph is unlikely more efficient than ordering a list in the demonstrated way. When regarding a cluster of tarps with a lot of overlaps, for each overlap there needs to be an edge connecting them. For n vertices there would be up to n^2 edges, making the graph data strucutre a lot more memory heavy than our list approach. In the graph all edges need to be considered for the shortest path, in lists only the closest tarp above is relevant, so the computation stops earlier.
Moreover, representing ranges on a tarp as vertices isn't a lot more useful than a list because those ranges will always be ordered one after another - just like in a list. Particularly, if we consider how water can move in this scenario, there are only two possible directions: vertically, by just dropping down, and horizontally, by flowing down a tarp. This seems best represented by a two-dimensional matrix because neighbouring ranges (that can be reached either vertically or horizontally) are also neighbours in the matrix. Such a two-dimensional matrix is best represented as a list of lists in Haskell. We achieve exactly that by using lists of tarps and tarps with a list of ranges assigned to each of them.
So, finally it seems that the presented solution is favourable against a graph orientated approach both structurally and considering efficiency.