Box Stacking Problem using Dynamic and Greedy Approach

Noor Hatem Hassan Anwer
Faculty Of Computer Science
Misr International University
noor2206987@miuegypt.edu.eg

Nourhan Amr Mohamed
Faculty Of Computer Science
Misr International University
nourhan2205080@miuegypt.edu.eg

Jomana Ayman Mohamed
Faculty Of Computer Science
Misr International University
jomana2206759@miuegypt.edu.eg

Nourhan Muhammed abdelzaher
Faculty Of Computer Science
Misr International University
nourhan2202382@miuegypt.edu.eg

Shaden Essam
Faculty Of Computer Science
Misr International University
shaden2208768@miuegypt.edu.eg

Ashraf AbdelRaouf
Misr International University
Professor in Computer Science
ashraf.raouf@miuegypt.edu.eg

***Abstract*—The box stacking problem is a well-known combinatorial optimization problem in the field of computer science and operations research.This problem focuses on creating the stack with the maximum height possible.In this paper, we focus on two main approaches to solve the box stacking problem: The dynamic programming approach and The greedy approach. Dynamic programming uses bottom-up approach by dividing the problem into smaller sub-problems and consider all possible combinations to find the optimal solution, On the other hand, Greedy makes quick decisions based on the current best option with the hope that these local optimizations will lead to a global optimum.**

***Index Terms*—Box stacking problem,Dynamic programming approach and Greedy approach**

I. Introduction

Box stacking problem is solved by, you are given N number of boxes, each box can have more than 1 instance that can be used, can be rotated 3 different times giving 3 different dimensions combinations. This problem involves arranging a this set of boxes, which may vary in size, weight, and shape, in a way that maximizes the height by taking into consideration that the base area of each lower box must exceed the base area of the box positioned above it.

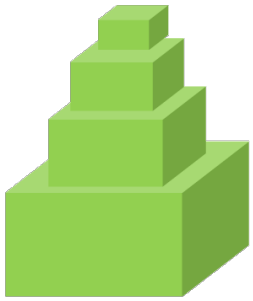

Fig. 1. Example of a the Box Stacking

II. Rotations of Boxes

We can have multiple instances of a box, which means we can rotate the box as we want and then use it as another box. We know there are six faces of a cuboid. We can have six instances of a box, but there will be only three unique instances, so indirectly, we will have 3*N boxes with different dimensions, Initially, the height of the box is designated as the original height. Then, the original width and original length are chosen, ensuring that the minimum dimension consistently corresponds to the length, while the maximum dimension consistently corresponds to the width ensuring that the base area of each box is larger than the base area of the box above it you create a uniform way to handle rotations. This consistency simplifies the logic required to check if a box can be placed on another.

$$\begin{aligned} &\text{box[index].height} = \text{Height[i]};\\ &\text{box[index].width} = \max(\text{Width[i]}, \text{Length[i]});\\ &\text{box[index].length} = \min(\text{Width[i]}, \text{Length[i]});\\ &\text{index}++; \end{aligned} \tag{1}$$

Example of the 1st rotation

III. Dynamic Programming Approach

How Does Dynamic Programming Work?

- Identify Sub-problems: Divide the main problem into smaller, independent sub-problems.
- Store Solutions: Solve each sub-problem and store the solution in a table or array.
- Build Up Solutions: Use the stored solutions to build up the solution to the main problem.
- Avoid Redundancy: By storing solutions, it ensures that each sub-problem is solved only once, reducing computation time.

A. Approach

Box stacking problem, utilizes a bottom-up approach to break down the main problem into smaller sub-problems.

This strategy involves considering all possible combinations of stacking boxes to find the optimal solution. First, we have to sort the boxes according to their areas in descending order then create an array to store the heights of all boxes `maxHeightt`

$$maxHeightt[i] = maxHeightt[j] + box[i].height; \quad (2)$$

When applying this equation,you have to take into consideration that the current height in the array `maxHeightt[i]` is less than any of the previous ones `maxHeightt[j]` + the current height of the box. Additionally, the length of `box[i]` must be strictly less than the length of `box[j]`, and the width of `box[i]` must be strictly less than the width of `box[j]`.

B. Algorithm

1) We applied rotations on each box.
2) We sorted the boxes in descending order according to the base area.
3) We created an array named `maxHeight` to store the height of each box and another array named `previous` to store the height of the previous box we selected.
4) We implemented nested for loops: the 1st loop (`i`) points to the upper box, and the 2nd loop (`j`) points to the box below `i`.
5) In the condition:
   Firstly, we check whether the width and length of box `i` are smaller than the width and length of box `j` (to ensure that the base area of the upper box is smaller than the box beneath it).
   Secondly, we check that the current height in the array `maxHeight[i]` is less than any of the previous ones `maxHeight[j]` plus the current height of the box.

C. Complexity analysis

Dynamic approach complexities:

- Time Complexity O(n^2)
- Space Complexity O(n)

Calculations for Time complexity:

- The for loop of the rotations represent O(n)
- sort function represent O(n log n)
- The for loop which fill the `maxHeightt` array with the heights of the boxes O(n)
- The nested for loop represent O(n^2)
- The for loop which compares the maxHeights represent O(n)
- printStack which prints the boxes represent O(n)
- The overall time complexity is the sum of all steps: O(n) + O(n log n) +O(n)+ O(n^2) + O(n)+O(n) = O(n^2).

IV. Greedy Approach

A greedy algorithm is a technique that makes the best local choice at each step in the hope of finding the global optimum solution. This approach can be efficient and straightforward, but it doesn't guarantee the best overall outcome for all problems.

A. Approach

Box Stacking using greedy approach, in each iteration we find the maximum stack by choosing the best box in this iteration according to this condition

$$i == 0 \quad || \quad (3)$$

$$box[i].width < prevWidth \&\& \; box[i].length < prevLength \quad (4)$$

(i==0) ensures that box[0] is always included in the stack. the rest of the condition ensures that length of `box[i]` must be strictly less than the length of `prevWidth`, and the width of `box[i]` must be strictly less than the width of `prevWidth` to ensure that the stack is strictly decreasing.

B. Algorithm

1) We applied rotations on each box.
2) We sorted the boxes in descending order according to the base area.
3) We declared 4 variables:
   -maxheight to store the max height of the stack globally
   -currentheight to store the current height when we stack new box locally
   -prevWidth used to store the previous width of each box and we initialized it with the dimensions of first box
   -prevLength used to store the previous length of each box and we initialized it with the dimensions of first box
4) We created a for loop where `i` iterates on each box
5) In the condition:
   -Firstly, (i==0) ensures that box[0] is always included in the stack.
   -Secondly,the rest of the condition ensures that length of `box[i]` must be strictly less than the length of `prevWidth`, and the width of `box[i]` must be strictly less than the width of `prevWidth` to ensure that the stack is strictly decreasing.
   -When the condition is true it adds the height of the box to the `currentheight(locally)` and updates the `maximumheight (globally)` with `currentheight` then we update the `prevlength` with the length of the box we chose and update `prevwidth` with the width of the box we chose.

C. Complexity analysis

Greedy approach complexities:

- Time Complexity O(n log n)
- Space Complexity O(n)

Calcultaions for Time complexity:

- The for loop of the rotations represent O(n)
- sort function represent O(n log n)
- The for loop which stack the boxes O(n)
- printStack1 which prints the boxes represent O(n)
- The overall time complexity is the sum of all steps: O(n) + O(n log n) +O(n)+O(n) = O(n log n).

TABLE I
COMPLEXITIES COMPARISON

The Approaches	Complexities	
	Time	***Space***
Dynamic Programming	O(n^2)	O(n)
Greedy	O(nlogn)	O(n)

V. COMPARISON

Dynamic programming approach is more accurate than the greedy approach in our Box Stacking problem but overall the complexity of the greedy approach is faster and easier than the dynamic programming approach here are few reasons why the dynamic approach is better:

A. Space Complexity

In dynamic programming approach it stores each solution in each sub problem.The space complexity of the dynamic approach is O(n). On the other hand, In the greedy approach it uses fewer data structures and temporary variables and the space complexity of the greedy approach is O(n).

B. Time Complexity

In dynamic programming approach it divides the problem into sub problems , the time complexity of the dynamic approach is O(n^2). On the other hand, The greedy approach makes the best local choice at each step in the hope of finding the global optimum solution, so the time complexity of the greedy approach is O(n log n).

C. Comparison summary

In the Box Stacking problem, regardless that the time complexity of the greedy approach is better and simpler, dynamic approach gives us more optimal solution and makes sure by changing the values of each box it will always return the best solution.

CONCLUSION

In conclusion, this paper compares the performance of box stacking using dynamic programming approach and greedy approach.The results show that the dynamic programming is more preferable due to its efficient utilization by dividing the problem into sub problems and ensures to provide the optimal solution.

REFERENCES

[1] https://www.geeksforgeeks.org/box-stacking-problem-dp-22/
[2] https://tutorialhorizon.com/algorithms/dynamic-programming-box-stacking-problem/
[3] https://favtutor.com/blogs/box-stacking-problem#: :text=Dynamic