Final Documentation for Box Sorting Algorithm

Omer Berk GENCER

Mirac Can YUKSEL

Contents

I. Problem Description

The Box Stacking problem involves trying to build a maximum height stack of boxes using dynamic programming techniques. It is a variation of the LIS(Longest Increasing Subsequence) problem. This particular instance of the problem only uses width and length, disregarding height. Following are the key points to note in the problem statement:

1) A box can be placed on top of another box only if both width and depth of the upper placed box are smaller than width and depth of the lower box respectively.

2) We can rotate boxes such that width is smaller than depth. For example, if there is a box with dimensions {2x3} where 2×3 is base, then there can be two possibilities, {2x3} and {3x2}

3) We can use multiple instances of boxes. What it means is, we can have two different rotations of a box as part of our maximum height stack.

II. Solution Description

1) Sort the boxes in decreasing order of base area. For optimal solution, heights for all boxes are considered equal at 1.

2) Define H, initially holding 1 for each box and R, holding the index of each box. Note that each index in both H and R corresponds to a box in a list storing boxes.

3) Compare every box with larger boxes in order to determine if stacking is possible.

4) If stacking is possible, and height achieved is greater than what was recorded at H for that box, increment H value and note the stacking pair in R.

5) After the iteration, the maximum value in H is the highest height and its index represents the smallest box of our stack. By following its value in R, we can find the next box of our stack. Following this gives us the solution from smallest to highest in box in our stack areawise.

III. Solution

```
from collections import namedtuple
from itertools import permutations
dimension <- namedtuple("Dimension", "length width")
FUNCTION sort_by_decreasing_area(rotations):
    RETURN sorted(rotations, key=lambda dim: dim.length * dim.width, reverse=True)
ENDFUNCTION

FUNCTION can_stack(box1, box2):
    IF box1.length < box2.length AND box1.width < box2.width:
        RETURN True
    ELSEIF box1.length < box2.width AND box1.width < box2.length:
        RETURN True
    ENDIF
ENDFUNCTION

FUNCTION box_stack_max_height(dimensions):
    boxes <- sort_by_decreasing_area(dimensions)
    num_boxes <- len(boxes)
    T <- [1 for rotation in boxes]
           ENDFOR
    R <- [idx for idx in range(num_boxes)]
             ENDFOR
    for i in range(1, num_boxes):
        for j in range(0, i):
            IF can_stack(boxes[i], boxes[j]):
                stacked_height <- T[j] + 1
                IF stacked_height > T[i]:
                    T[i] <- stacked_height
                    R[i] <- j
            ENDIF
                ENDIF
    ENDFOR
        ENDFOR
    max_height <- max(T)
    start_index <- T.index(max_height)
    while True:
        OUTPUT boxes[start_index]
        next_index <- R[start_index]
        IF next_index = start_index:
            break
        ENDIF
        start_index <- next_index
    ENDWHILE
    RETURN max_height
ENDFUNCTION
```

O(n)

O(n)

O(n)

O(n)

O(n²)

O(n)

O(n²)

IV. Time Complexity Analysis

When analyzing the Big-O performance of sorting algorithms, n typically represents the number of elements that you're sorting.

The time complexity of the program is $O(n^2)$ since program only has one nested loop with dept of 2 and no other section with a time complexity larger than O(n).

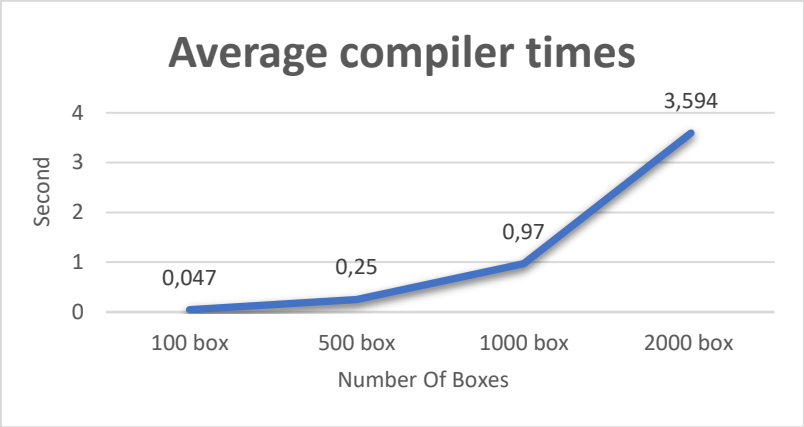
Average compiler times:

With 100 boxes: 0.047s

With 500 boxes 0.250s

With 1000 boxes 0.970S

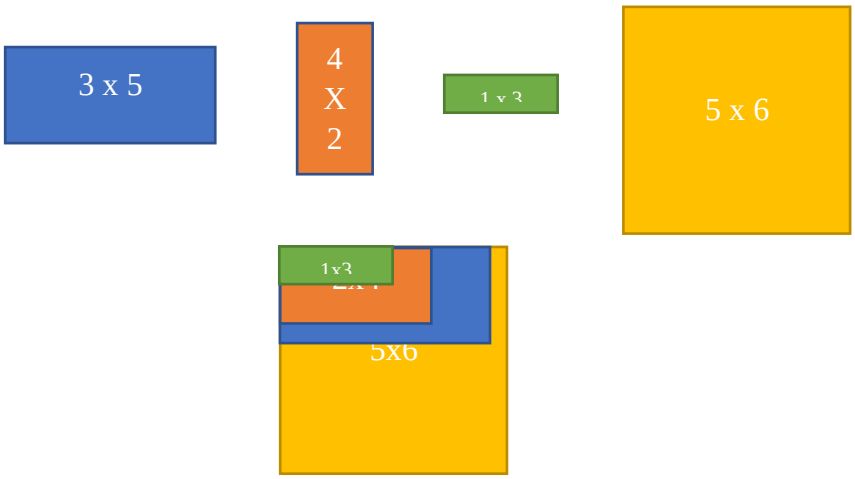
With 2000 boxes 3.594S



V. Input & Output Description

Input:

[(3, 5), (4, 2), (1, 3), (5, 6)]



Output:

Number of Boxes:

4

Order:

[(1,3), (2,4), (3,5), (5,6)]

VI. List of modification

- **run():** A function named run has been added in order to deal with user input. Function tries to access a file named sizes.txt in the same folder as the code. If this txt file is not present, it instead asks for the user to input sizes.
- In input and output format has been changed. On final version inputs are write comma between two digits and outputs printing horizontally rather than vertically.
- Added new input method. Program can accept input from file.
- **General Improvements:** Several small improvements were made to eliminate potential bugs.
- **Test Cases:** Test cases has been added.

VII. User Manual

Program was written in Python version 3.7. It takes a list of named tuples with two elements corresponding to height and width. Any number of boxes can be given. To input data to program from a file, user should create a txt file called sizes.txt on the same folder as the program. If this file is not given, program will ask the user to manually input data. This input should be of the form “X, Y” where X and Y can be integer or float, or a combination of both. User can type ‘f’ to stop inputting data.

box_stack_max_height() - takes the list of boxes and outputs maximum stack.

can_stack() - checks the stickability of two boxes with both rotations.

sort_by_decreasing_area() - sorts the list of boxes by their area.

VIII. Test Descriptions and Conclusions

- **test_int:**

data = ("2,3")

This test ensures that program can successfully take integer values. This test is successful and program behaves as expected.

- **test_float:**

data = ("2.6,3575.4646")

This test ensures that program can successfully take float values. This test is successful and program behaves as expected.

- **test_intfloat:**

data = ("2,3.7")

This test ensures that program can successfully take a combination of float and integer values. This test is successful and program behaves as expected.

- **test_wrong_type:**

data = ("2.5,'hello'")

This test ensures that program will not crash and display a warning upon receiving an incorrect data. This test is successful and program behaves as expected.

- **test_empty:**

data = (' ')

This test ensures that program will not crash and display a warning upon receiving an empty data. This test is successful and program behaves as expected.

IX. Job Partition

Algorithm Design – Omer, Mirac

Implementation & Test – Mirac

Documentation - Omer