Texture Packing

AuthorNames
(temporary vacancy during anonymous peer review)

Date: 2020-05-09

Chapter 1: Introduction

1.1 Background of the Problem

The Texture Packing problem is to pack multiple rectangle shaped textures into one large texture. The resulting texture must have a given width and a minimum height. The problem comes from the two-dimensional packing problem. It's a complicated combinatorial optimization problem. Packing problem is a "*NP-HARD*" problem. The classic packing problem requires placing a certain number of items in some boxes of the same size so that the sum of the items in each box does not exceed the size of the box and the number of boxes used is minimal. Packing problems can be divided into three types: one-dimensional packing problem, two-dimensional packing problem and three-dimensional packing problem.

The research of two-dimensional rectangle packing problem we want to study is to combine the small picture into a large one and load it into memory as a large texture in practical application, so as to achieve the purpose of reducing memory consumption. It's worth noting that anyone who works in computer game design knows the texture packer tool, which works in a similar way.

1.2 Description of the Problem

What we need to do is to design an approximation algorithm to obtain

an optimal solution to the two-dimensional fixed-width strip packing problem. Our goal is to load a certain number of rectangles into a box of a fixed width and get as little height as possible, that is, take up less space, as shown in the figure below. Assuming that five rectangles are put into the box, the result on the right side is obviously better than that on the left side.

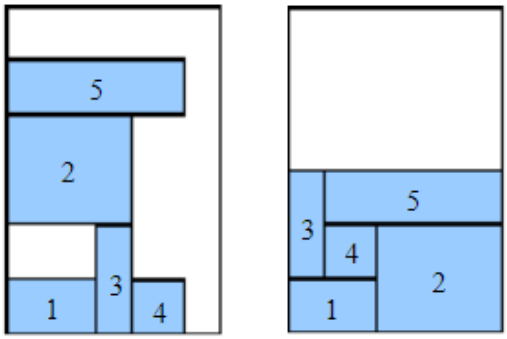

Diagram 1: Problem Diagram

1.3 Project Requirements

In this project we are required to design an approximation algorithm which has a polynomial time complexity. And then we randomly generate some test cases of different sizes (from 10 to 10,000) with different distributions of widths and heights. And finally we take all factors that might affect the approximation ratio of our proposed algorithm to do a thorough analysis.

Chapter 2: Data Structure / Algorithm Specification

In the chapter, we will briefly discuss the level-oriented algorithms

from Trojan[1]: the Brute-Force and FFDH algorithms. Firstly we will analyze the problem: related concepts, algorithm design and several bounds.

2.1 Analyze the Problem

The packing algorithms that we analyze in this section both assume that the rectangles in the list L are ordered by decreasing(actually, nonincreasing) height, and they pack the rectangles in the order given by L. So as to form a sequence of levels. All rectangles will be placed with their bottoms resting on one of these levels. The first level is simply the bottom of the bin. Each subsequent level is defined by a horizontal line drawn through the top of the first (and hence maximum height) rectangle placed on the previous level. Notice how this corresponds with one-dimensional bin-packing; the horizontal slice determined by two adjacent levels can be regarded as a bin(lying on its side) whose width is determined by the maximum height rectangle placed in that bin. The following two level algorithms are suggested by analogous algorithms studied for one-dimensional bin-packing.

2.2 The Brute-Force Algorithm

In the algorithm, we just sort all rectangles by height in decreasing order and then arranges them in order from left to right to fill the space. If more rectangles can not be put down, a new row will be created to accom-

modate it.

The pseudo-code:

```
# Sort by height
Sort(rectangles[], width[])
# Arrange in order
for i in range(0,n):
    if (width of the rectangle+occupied width <= width):
        # The current line can fit
        arrange it;
    else:
        # Create a now row
        create a new row;
        arrange it;
printResult(rectangles, arrangement, width)
```

2.3 The FFDH Algorithm

In the algorithm, we just sort all rectangles by height in decreasing order and then arranges them to the first row with enough width to fit them. If more rectangles can not be put down, a new row will be created to accommodate it. So we call it "*First-Fit Decreasing-Height*" (FFDH) algorithm.

Theorem 1[1], [2]: For any list L ordered by nonincreasing height:

$$\mathrm{FFDH}(L) \leq 1.7 \cdot \mathrm{OPT}(L) + 1$$

where $\mathrm{OPT}(L)$ is the minimum possible bin height with in which the rectangles in L can be packed and $\mathrm{A}(L)$ denote the height actually used by a particular algorithm when applied to L.

The pseudo-code:

```
# Sort by height
Sort(rectangles[], width[])
# Arrange in order
for i in range(0,n):
    # Find the first row that can fit it.
    for j in range(0, tot+1):
        if (width of the rectangle+occupied width <= width):
            find the first row that can fit it.
            break
    # Exist no row to fit it.
    if (flag == 0):
        create a new row to fit it.
printResult(rectangles, arrangement, width)
```

2.4 Factors Affecting Approximate Efficiency and Improvements

The two key factors affecting the approximate efficiency of our algorithm are the Average Ratio of the Length to Width (ARLW) of the rectangles and the rectangle number. However, there exists some improvements to our algorithms.

The level-oriented packing methods analyzed here would seem to be wasteful of space, since they never consider packing rectangles in the space between the tops of the shorter items in a block and the top half of the block itself. Thus one might expect that the non-level-oriented Bottom-Left methods of [2] which pack by always placing the next rectangle from L as low as possible and then as far to the left as possible, would perform significantly better.

The first of possible improvement is that, instead of packing every block in a left-to-right manner, one might alternately pack blocks from left-to-right and then from right-to-left. Then the tallest rectangle in the block will

be above the shortest, rather than the tallest, rectangle in the preceding block, and some reduction in total height might be achieved by dropping each rectangle until it touches some rectangle below it. Another approach would be to allow more general packings within blocks, say by allowing new, shortened blocks to be created in the space above the regular items in a block, which otherwise would remain empty. However, the essentially one-dimensional nature of the worst case examples mentioned in the paper show that the asymptotic performance bound scan not be improved by these modifications, and so one can only hope that they will do better in practice. One possible modification of our basic model deserves mention. Suppose that, in packing, we are allowed to rotate rectangles by 90 if we so wish.

Chapter 3: Testing Results

3.1 Test Environment

Our test environment is just as following:

OS: Windows 10 Pro for Workstations, Build 18363.815

Compiler: Python 3.8.2

Compile command: python source.py

RAM: 64GiB

3.2 Test Results and Analysis

We use three input to test the correctness of the two algorithms. It lies in the /data/test_correction[0-2]/data.in folder. The results of the two algorithms are result1.jpg and result2.jpg. The result agrees with our expectations. Hence the algorithms are correctly implemented.

Then we use seven input to test the time complexity of the program. It lies in the /data/test[0-7]/data.in folder. The result agrees with our expectations. Hence the algorithms are correctly implemented.

Chart 1: Time-Number Complexity Chart

Algorithm	Program1		Program2	
N	ans	time	ans	time
10	62	0.00000	60	0.00000
100	2009	0.00100	1734	0.00100
500	132685	0.00100	107098	0.00100
1000	530407	0.02593	419428	0.07979
3000	4895144	0.02393	3838961	0.55950
5000	15274026	0.06782	11853300	1.22641
7000	3771186	0.06283	2931069	2.47342
10000	50938186	0.12068	39410293	4.02726

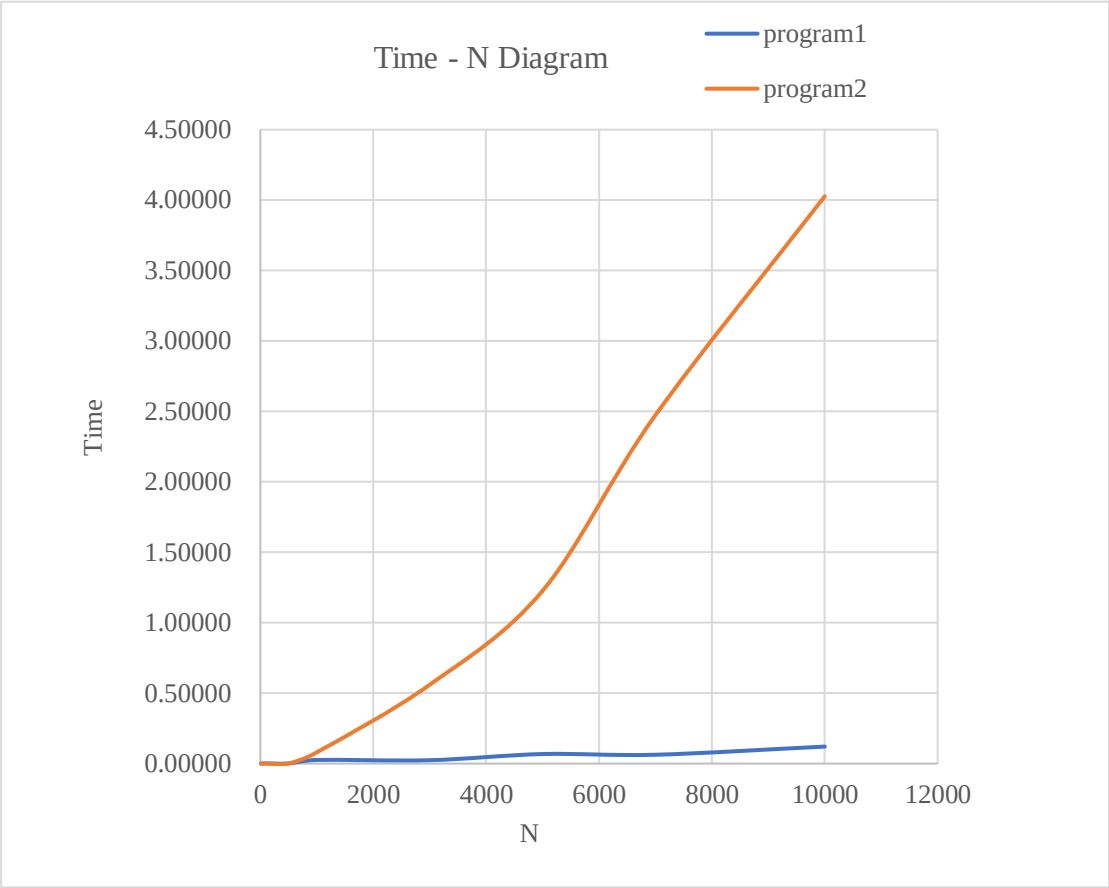

Diagram 2: Time-Number Complexity Diagram

Chapter 4: Analysis and Comments

4.1 Time Complexity Analysis

The time complexity for the two algorithms are $O(N \log N)$ and $O(N^2)$.

In the first algorithm, we need $O(N)$ to read the rectangles and $O(\log N)$ to arrange it. And by the same reason it's $O(N^2)$ for the 2nd algorithm.

4.2 Space Complexity Analysis

The space complexity for the two algorithms are simply $O(N)$ to store the rectangles.

Appendix: Source Code

The source code folder consists of two files: 1.py and 2.py corresponding to algorithm 1 and algorithm 2.

The 1.py:

```
import matplotlib.pyplot as plt
import time

'''
    This part is the visualization part, which
    displays the results in the form of images.

    params
        rect: A list of rectangular lengths and
              widths
        loca: A list of coordinates in the lower
              left corner of a rectangle
        width:A width limit given by the input

'''
def printResult(rect, loca, width):

    # A list is used to store some selected colors
    mycolor = ["aquamarine", "dodgerblue", "rosybrown", "maroon", "khak
i", \
        "mediumpurple", "cornflowerblue", "slategrey", "tan", "salmon"]
    color_num = len(mycolor)

    # Set the size of the canvas based on the output
    # rectangle collection
    mxa = 0
    mxb = 0
    for i in range(0, len(rect)):
```

```
            mxa = max(rect[i][0]+loca[i][0], mxa)
            mxb = max(rect[i][1]+loca[i][1], mxb)

        fig = plt.figure(figsize=(4, 8))

        ax1 = fig.add_subplot(111)
        ax1.set_xlim([0, width])
        ax1.set_ylim([0, 1.6*mxb])

        # Print rectangle on frequency screen
        cur = 0
        for i in range(0, len(rect)):
            tmp = plt.Rectangle((loca[i][0], loca[i][1]), rect[i][0], rect[
i][1], \
                facecolor=mycolor[cur], edgecolor="white")
            ax1.add_patch(tmp)
            cur = (cur+1)%color_num

        plt.savefig("result1.png")

    '''
        Main part. The algorithm used is an approximate
        algorithm, which sorts all rectangles according
        to the height as the key word from large to small,
        and then arranges them in order from left to right.
        If more rectangles cannot be put down, a new row
        will be created.
    '''
    if __name__=='__main__':


        # input data
        # rect[] stores the length and width of the rectangle,
        # and loca[] stores the coordinates of the lower left
        # corner of the rectangle.
        rect = []
        loca = []

        n,width = list( map(int, input().split() ) )
        start = time.time()
        for i in range(0,n):
            rect.append( list( map(int, input().split() ) ) )

        # Sort by height
        rect.sort(key=lambda x:x[1], reverse=True)
```

```
    loca.append([0, 0])
    for i in range(1,n):
        loca.append([loca[i-1][0]+rect[i-1][0], 0])

    # Arrange in order
    ans = 0
    loca[0][0] = loca[0][1] = 0
    curx = 0
    cury = 0
    tmp = rect[0][1]
    for i in range(0,n):
        if (curx+rect[i][0] <= width): # The current line can fit
            loca[i][0] = curx
            loca[i][1] = cury
            curx = curx + rect[i][0]
        else:                          # Create a now row
            curx = 0
            cury = cury + tmp
            tmp = rect[i][1]
            loca[i][0] = curx
            loca[i][1] = cury
            curx = curx + rect[i][0]
            ans = max(ans, cury+tmp)
    end = time.time()
    print("The height of program1 is: ",ans)
    print("The time of program1 is: ",end-start)
    print('The rectangles and their locations:')
    print(rect)
    print(loca)
    print('\n')
    printResult(rect, loca, width)
```

The 2.py:

```
import matplotlib.pyplot as plt
import time

'''
    This part is the visualization part, which
    displays the results in the form of images.

    params
        rect: A list of rectangular lengths and
              widths
        loca: A list of coordinates in the lower
```

```
            left corner of a rectangle
        width:A width limit given by the input

'''
def printResult(rect, loca, width):

    # A list is used to store some selected colors
    mycolor = ["aquamarine", "dodgerblue", "rosybrown", "maroon", "khak
i", \
        "mediumpurple", "cornflowerblue", "slategrey", "tan", "salmon"]
    color_num = len(mycolor)

    # Set the size of the canvas based on the output
    # rectangle collection
    mxa = 0
    mxb = 0
    for i in range(0, len(rect)):
        mxa = max(rect[i][0]+loca[i][0], mxa)
        mxb = max(rect[i][1]+loca[i][1], mxb)

    fig = plt.figure(figsize=(4, 8))

    ax1 = fig.add_subplot(111)
    ax1.set_xlim([0, width])
    ax1.set_ylim([0, 1.6*mxb])

    # Print rectangle on frequency screen
    cur = 0
    for i in range(0, len(rect)):
        tmp = plt.Rectangle((loca[i][0], loca[i][1]), rect[i][0], rect[
i][1], \
            facecolor=mycolor[cur], edgecolor="white")
        ax1.add_patch(tmp)
        cur = (cur+1)%color_num

    plt.savefig("result2.png")



'''
    Main part. The algorithm used is an approximate
    algorithm, which sorts all rectangles according
    to the height as the key word from large to small,
    and then arranges them to the first row that can fit
    it. If more rectangles cannot be put down, a new row
```

```
    will be created.
'''

if __name__=='__main__':

    # input data
    # rect[] stores the length and width of the rectangle,
    # and loca[] stores the coordinates of the lower left
    # corner of the rectangle.
    rect = []
    loca = []

    n,width = list( map(int, input().split() ) )

    start = time.time()
    for i in range(0,n):
        rect.append( list( map(int, input().split() ) ) )

    # Sort by height
    rect.sort(key=lambda x:x[1], reverse=True)
    loca.append([0, 0])
    for i in range(1,n):
        loca.append([loca[i-1][0]+rect[i-1][0], 0])

    # Arrange in order
    loca[0][0] = loca[0][1] = 0
    curx = []
    cury = []
    tot = -1
    tmp = 0
    ans = 0
    for i in range(0,n):
        flag = 0
        # Find the first row that can fit it.
        for j in range(0, tot+1):
            if (curx[j]+rect[i][0] <= width):
                loca[i][0] = curx[j]
                loca[i][1] = cury[j]
                curx[j] = curx[j] + rect[i][0]
                flag = 1
                break

        # Otherwise create a new row
        if (flag == 0):
```

```
            tot = tot+1
            curx.append(0)
            cury.append(tmp)
            if (tot >= 1):
                cury[tot] = cury[tot] + cury[tot-1]
            tmp = rect[i][1]
            loca[i][0] = curx[tot]
            loca[i][1] = cury[tot]
            curx[tot] = curx[tot] + rect[i][0]
            ans = max(ans, cury[tot]+tmp)

    end = time.time()

    print("The height of program2 is: ",ans)
    print("The time of program2 is: ",end-start)
    print('The rectangles and their locations:')
    print(rect)
    print(loca)
    print('\n')
    printResult(rect, loca, width)
```

References

[1] Coffman, Jr EG, Garey MR, Johnson DS, Tarjan RE. Performance bounds for level-oriented two-dimensional packing algorithms. SIAM Journal on Computing. 1980 Nov;9(4):808-26.

[2] Baker BS, Coffman, Jr EG, Rivest RL. Orthogonal packings in two dimensions. SIAM Journal on computing. 1980 Nov;9(4):846-55.

Author List

(temporary vacancy during anonymous peer review)

Declaration

We hereby declare that all the work done in this project titled "Texture Packing" is of our independent effort as a group.

Signatures

(temporary vacancy during anonymous peer review)