Project Five: Texture Packing

Date: 2020-05-02

# Chapter 1: Introduction

### Problem Description

Texture Packing is a strip-packing problem and is a common topic explored in the area of Approximation Algorithms. Unlike the *Bin Packing problem*, there are two parameters (width and height) instead of one (bin capacity) are taken into calculation.

Given a set of rectangles with dimensions width and height where , we are expected to pack them into a larger shape with a pre-specified width . We want to pack as many rectangles as possible into the larger one with the objective of minimizing the height, in polynomial time. Here, we will demonstrate how this can accomplished by using a the *Next-Fit Decreasing Height (NFDH)* approximation algorithm.

### Input and Output Specification

* Input

The program first receives, in one line, two positive integers and , where is the width of the texture atlas that the images will be packed into and is the number of images to be packed. The program then receives lines of input. Each line specifies an image and contains two positive integers and , the pixel height and width of the image, respectively.

* Output

The program outputs lines, specifying the positions of the images in the texture atlas. Each line contains two non-negative integers and , the and position of the bottom-left pixel of the image in the texture atlas, respectively. The positions are outputted in the same order as the input sizes are input; that is, the position in output corresponds to the image in input.

# Chapter 2: Algorithm Specification

* 1. Data Structures
* Min Heap

A min heap is used in the implementation of heap sort, and the resulting sequence is stored into an array.

* 1. Algorithm Specifications
* The Main Idea

The NFDH algorithm is an off-line algorithm, where all input is considered before an output is produced. Every inputted rectangle is sorted by their height in decreasing order, where the tallest gets dealt with first and is placed in the left of the texture atlas. A *next-fit* approach is used to pack the shapes, where a rectangle is packed in if and only if it does not exceed the width requirement. If the next rectangle to be packed goes over the width, it will be placed above the previous shape and justified left. This algorithm tends to pack the rectangles level-by-level, and earlier levels cannot be accessed.

* Sorting the Input- buildheap(), sort(), pop()

To sort the inputted rectangles in decreasing order by their height, we use heap sort. First, we build a min-heap using the address of each rectangle as the node’s key. We then pop the root, place it into the array node[] and heapify accordingly.

# Chapter 3: Testing Results

* 1. Test Cases

Random Case:

-Random heights and widths, from [0, width of bin]

Diagonal Case:

-Each item has Height == Width

Mismatched Diagonal Case:

-Each item has Width of item = Width of Bin – Height of item

Best Case:

-Smallest size: n == 10

-All items can fit together in the same bin

Worst Case:

-Largest size: n == 10000

-No items can fit together in the same bin

All-Pairs Case:

-All items are fixed height, fixed bin width=10

-1/3 of items are length 4, 1/3 are length 6, rest of items (1/3+1) are length 5

-The optimal solution would have no whitespace

-Every bin would look like one the bins below:

A screenshot of a cell phone

Description automatically generated

Filled (No Gaps) Case:

-In an optimal solution, every bin would look like one of these 4:

A screenshot of a cell phone

Description automatically generated

-In an optimal solution no bin would have any whitespace

Filled (With Gaps) Case:

-In an optimal solution, every bin would look like this:

A close up of text on a white background

Description automatically generated

-In an optimal solution every bin would have a 1x1 square of whitespace

Trivial Case:

-All items are the exact width of the bin

-Height==Width for every item

* 1. Correctness Testing

We know that for an optimal case, our filled, no gaps case should take up exactly 4 bins for every series of 12 items, and our filled, with gaps case should take up exactly 1 bin for every series of 4 items. We also know that our trivial case and worst case should take up exactly as many bins as the number of items in the case, and that our best case should take up exactly 1 bin. As well, we know that our all-pairs case has an optimal solution in which there is no whitespace, and given an even n will have exactly floor(n/3) + ceil(n/3) bins, which for n=1000 will be 500 bins.

We test these in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test case description** | Number of items (n) | # of bins used | Expected # of bins used |
| Filled (no gaps) case | 60 | 22 | 20 |
| Filled (with gaps) case | 20 | 7 | 5 |
| Trivial Case | 100 | 100 | 100 |
| Worst Case | 10000 | 10000 | 10000 |
| All-Pairs Case | 1000 | 500 | 500 |

As we can see, all the amounts of bins used are either the optimal case, or within the approximation ratio (2 for NFDH) of the optimal case. Therefore, we know our algorithm is correct!

* 1. Performance Testing

-Time taken is accurate to 4 decimal places

|  |  |  |
| --- | --- | --- |
| **Test case description** | **Number of items (n)** | **Time taken (in ms)** |
| Best Case | 10 | 0.088 |
| Filled (with gaps) Case | 20 | 0.125 |
| Filled (w/ no gaps) Case | 60 | 0.22 |
| Trivial Case | 100 | 0.3287 |
| Random Case | 500 | 1.4937 |
| All-Pairs Case | 1000 | 2.491 |
| Diagonal Case | 3000 | 9.7453 |
| Mismatched Diagonals Case | 4000 | 13.167 |
| Random Case | 5000 | 16.6687 |
| Random Case | 7000 | 23.08 |
| Random Case | 9000 | 29.8653 |
| Worst Case | 10000 | 32.853 |

The graph made from these test cases is show below:

![A screenshot of a cell phone

Description automatically generated]()

In this graph, the black line (in the very back) shows the line of best fit, done via linear regression, the red dots are the points that we tested, and those points are connected with the dotted cyan line. As you can see the cyan line only differs slightly from the best fit line, while the best fit line is close or equal to O(n). This is because for smaller n, O(nlogn) and O(n) are very close, and as seen in the dip at the beginning of the graph, for very small n O(nlogn) can be even faster. As the number of items increase, however, the cyan line moves ever so slightly above the black line, showing that for larger n the program does indeed break the O(n) upper bound. We believe the reason we cannot see a significant breaking of this bound is only due to the upper limit of test case sizes given in the question, which is 10000.

Running the worst case on my personal computer, however, already takes up to ~35 seconds in real time, since the real bound of the program is more so gcc’s scanf implementation than any part of our algorithm (which was not counted for our tests), so trying to find the n for which the algorithm would more cleanly break an O(n) upper bound would be impractical.

# Chapter 4: Analysis and Comments

* 1. Time Complexity
  2. Space Complexity
  3. Approximation Scheme

# Conclusion

# Appendix: Source Code

# Declaration

*We hereby declare that all the work done in this project titled “Project Five: Texture Packing” is of our independent effort as a group.*

# Duty Assignments

**Programmer:**

**Tester:**

**Report Writer:**

# References