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.

10 10

4 5

1 1

3 2

1 6

2 9

1 3

2 2

2 4

5 3

8 1

* Output

The program first outputs the total area used by the texture atlas. It then outputs lines, specifying the positions of the images in the texture atlas. Each line contains two positive 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.

area=130

(3, 0)

(8, 12)

(7, 9)

(2, 0)

(0, 0)

(9, 0)

(5, 9)

(7, 0)

(0, 9)

(0, 12)

# 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. A struct was used to represent this data structure and had attributes each corresponding to the nodes, number of nodes in the heap, and heap capacity.

*struct* minheap\_struct{

*int* \*nodes;

*int* size;

*int* capacity

};

Note that the node does not store the height of each inputted rectangle, but rather the ***address*** of each shape.

* 1. Algorithm Specifications
* Next-Fit Decreasing Height (NFDH) Algorithm: 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.

* Packing the Rectangles Together – nfdhPack()

The pseudocode uses both rectangles and images interchangeably. For reference purposes, images will refer to the inputted set of rectangles.

*int* nfdhPack(){

*int* lvl\_height; //lvl\_height stores the height of each level

*int* used\_width; //used\_width stores width of space already occupied in each level

/\*empty level implies lvl\_height = 0 and used\_width = 0.\*/

/\*Initializing pointer variables;

current\_image is initialized to the rectangle with the greatest height\*/

    current\_lvl := 0; //Points to the top or “current” level

    current\_image := images[0]; //Points to the current rectangle to be packed

    current\_y := 0; //Contains the current y-coordinate of the rectangles

/\*Initialize variables to hold coordinates of the current rectangle being worked with;

The coordinates are defaulted to the bottom left (0,0)\*/

x\_pos[current\_image] := 0;

y\_pos[current\_image] := 0;

/\*Initialize the attributes of the first level with the height and width of

the first rectangle\*/

    lvl\_height[0] := image\_height[current\_image]; //stores the height of each level

    used\_width[0] := image\_width[current\_image]; //stores width of space occupied per level

    for each rectangle to be packed:

        current\_image := current rectangle to be packed;

/\*Check if the current rectangle can be packed into the texture atlas\*/

        if (width of the level of current\_image > remaining width of texture atlas

|| height of current level < height of current rectangle):

            current\_y += add the height of the previous level

            set current\_lvl to the next level;

            lvl\_height[current\_lvl] := height of the current rectangle;

/\*Update coordinates of packed shape in texture atlas\*/

            x\_pos[current\_image] := left-justified position of current rectangle and level;

            y\_pos[current\_image] := justified position of current rectangle and level;

            update remaining amount of width in current level;

    current\_y := current y-coordinate + height of the current level

    return current\_y;   //Total height of the texture atlas is returned

}

The above pseudocode is an implementation of the NFDH approximation algorithm. As described in “*Next-Fit Decreasing Height (NFDH) Algorithm: The Main Idea”*, the idea is to take a pre-sorted set of rectangles and pack them into a texture atlas, creating a new level each time the specified width is exceeded. Our implementation first takes the rectangle with the greatest height and initializes the current height and width of the first level. For every rectangle after the first rectangle, the algorithm first checks to see whether the current level is too short or if there is too little unused width remaining to fit the shape. If so, the rectangle gets packed into the atlas at the next level. This in turn makes the next level, updates the y-coordinate of where that rectangle is placed and the height of the new level. Else, we can pack the next rectangle in the current level. Regardless of whether the rectangle gets packed into a new level, the x and y coordinates, and the current width of the current level gets updated. Finally, the total height of the texture atlas gets returned.

# Chapter 3: Testing Results

* 1. Test Cases

Let be the size of the set of the rectangles to pack,be the width of each level, and respectively be the width and height of each rectangle where .

* *Random Case:* Heights and widths of the rectangles are randomized, ranging from
* *Trivial Case:* All rectangles have the same width and height of .
* *Diagonal Case:* All rectangles have the same width and height.
* *Mismatched Diagonal Case:* Each rectangle has .
* *Best Case:* Smallest size of , where all 10 rectangles must fit together in the same level.
* *Worst Case:* Largest size: , where no rectangle can fit together in the same level.
* *All-Pairs Case:* All items have a fixed , where , such that , and a fixed . The optimal solution would have no whitespace, pictured below.

A screenshot of a cell phone

Description automatically generated

* *Filled (No Gaps) Case:* In an optimal solution, no level would have any whitespace, and would look like either or :

A screenshot of a cell phone

Description automatically generated

* *Filled (With Gaps) Case:* In an optimal solution, every level would have a unit of whitespace, depicted below.

A close up of text on a white background

Description automatically generated

* 1. Correctness Testing

We know that for an optimal case, our *filled*, *no gaps case* should take up exactly 4 levels for every series of 12 rectangles, and our *filled*, *with gaps case* should take up exactly 1 level for every series of 4 rectangles. We also know that our *trivial case* and *worst case* should take up exactly as many levels as the number of rectangles in the case, and that our best case should take up exactly 1 level. As well, we know that our *all-pairs case* has an optimal solution in which there is no whitespace, and given an even will have exactly levels, which for will be 500 levels. We test these cases in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test case description** | Number of rectangles () | # of levels used | Expected # of levels 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 levels used are either the optimal case, or within the approximation ratio of 2 () for the optimal case. Therefore, we know our algorithm is correct!

* 1. Performance Testing

|  |  |  |
| --- | --- | --- |
| **Test case description** | **Number of items ()** | **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 |

\*Time taken is accurate to 4 decimal places.

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 is the line of best fit, done via linear regression, the red dots are the points that we tested, and those points are connected with a dotted cyan line. The cyan line only differs slightly from the best fit line, while the best fit line is close or equal to . This is because for smaller , and are very close, and as seen in the dip at the beginning of the graph, for very small , can be even faster. As the number of rectangles increase, however, the cyan line moves ever so slightly above the black line, showing that for larger the program does indeed break the 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 takes up to 35 seconds in real time, and 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), trying to find the for which the algorithm would more cleanly break an upper bound would be impractical.

# Chapter 4: Analysis and Comments

* 1. Time Complexity

As stated above in *Chapter 3*, our testing and analysis tends to a runtime of . For a more generalized analysis, we break apart our algorithm into three key components – buildHeap(), sort() and nfdhPack(). The cost of pointers and malloc/calloc calls are omitted in this calculation.

The algorithm to construct a min-heap of the rectangles according to their heights, buildHeap() takes time to do so. To sort the heap in decreasing order by their heights, heap sort (sort()) is used and finishes the task in time. Lastly, the running time of the NFDH algorithm (nfdhPack()) is bounded by , since the rectangles in the set are already pre-sorted.

Thus, the running time of our algorithm is

* 1. Space Complexity

The space complexity for our approximation algorithm is . Only one-dimensional arrays are used to keep track of the input, the x and y coordinates, heights of each level in the texture atlas and so on. Furthermore, the min-heap data structure uses an array implementation making it extremely efficient. Dynamic memory allocation is also utilized, which contributes to the effective use of space in the program.

* 1. Approximation Scheme

# Conclusion

Comparing the NFDH approximation algorithm to other strip-packing algorithms, it does not necessarily have the best performance or approximation ratio. Despite it being relatively naïve, it efficiently generates usable packings for almost all cases. However, the algorithm has clear flaws.

The algorithm tends to leave a large amount of empty space on the right side of each level. Algorithms exist which capitalize on this flaw, such as Baker’s Up-Down algorithm. Furthermore, in the case where one image is significantly taller than all the others, the level sorting approach leads to wasted space because a single image cannot span multiple levels. This problem can be solved by a wholistic packing algorithm, one that does not pack into levels.

When seeking a packing of images that reduces area, the problem as posed in this project is not sufficient. The width of the final packing is restricted to the value given in the input. It is possible that no truly efficient packing using this width can exist for a given set of images. In future research, 2-dimensional packing algorithms that can produce packings with a range of widths must be considered.

# 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