Texture Packing

Date: 2024-12-2

目录

Chapter 1: Introduction

本项目研究的主题是`Textur Packing`。Texture Packing 问题是一类矩形装箱问题，常用于计算机图形学和游戏开发领域。其目标是将多个矩形形状的纹理 (texture) 紧密地打包到一个更大的矩形纹理中。具体来说，题目要求如下：

1. **输入**：若干矩形纹理，每个纹理有固定的宽度和高度。
2. **输出**：一个大的矩形纹理，其宽度固定，要求设计一种方法尽量减少所需的高度。
3. **限制条件**：大矩形纹理的宽度是给定的，最终纹理必须尽可能紧凑。

由于题目没有明确给出输入输出，在此介绍我们项目当中期望的输入输出格式：

Input Specification(example)

```
10
5
4 3
3 2
6 5
5 3
7 1
```

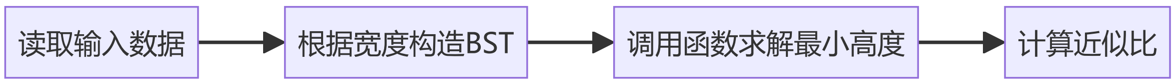
> 首先读取的两个整型数字分别是大纹理（容器）的最大宽度以及小纹理（小矩形）的个数`n`；
>
> 接着为`n`行的小矩形数据，宽x高的格式依次输入。

Output Specification(example)

```
...
---------------------conclusion---------------------
- Approximation Ratio for ceiling_height packing: 128.5714%
- Final height dealing with 5 rectangles by ceiling_height packing algorithm: 9
```

> 1. `...`表示的是依次插入小矩形的过程信息；
> 2. 最后分别输出`Approximation Ratio`与最终高度。其中`Approximation Ratio`采用算法得到的最终高度/最优解法的理想高度`optimal_height`，后者通过`area_sum/container_width`得到，即将所有的矩形面积除以容器宽度。

- **主要流程**



Chapter 2: Data Structure / Algorithm Specification

2.1 结构体介绍

2.1.1 Texture

```
typedef struct {
    int width;
    int height;
} Rectangle;

// BST node structure
typedef struct TreeNode {
    Rectangle rect;
    struct TreeNode *left;
    struct TreeNode *right;
} TreeNode;
```

1. 为了表示存储单个`Texture`(下称小矩形)，我们定义结构体`Rectangle`，其内部属性分别为整形的宽与高;
2. 然后，我们将小矩形作为`BST`的节点根据属性`width`构造二叉树，方便后续根据宽度快速寻找符合条件的小矩形;

2.1.2 ceiling_height

```
#define MAX_WIDTH 1000
int ceiling_height[MAX_WIDTH];
```

`ceiling_height`记录容器单位宽度的各个最高高度。换言之，将大矩形容器内部的小矩形存放结果的最高值取出，各个单位宽度上的最高值作为`ceiling_height`的数组元素。

- **相关操作**
    - 每次搜索当前最低高度的连续宽度`consist_width`;
    - 在BST当中选择合适的小矩形插入到当前的位置，同时更新`ceiling_height`数组;
    - 如果在第二步当中无法找到适合的小矩形，将该`consist_width`宽度设置为相邻区域的较小高度（表示弃用），然后继续搜索。

> 具体实现在`2.2`中给出。

2.2 主要函数介绍

2.2.1 BST相关

- **创建与插入节点**

```
Function CreateNode(Rectangle rect) Returns TreeNode*
    Create a new TreeNode object
    If memory allocation fails
        Output "Memory allocation failed!" error message
        Exit the program
    Set the node's rectangle to the provided rect
    Set the node's left and right children to NULL
    Return the newly created node
End Function
```

```
Function Insert(root, Rectangle rect) -> TreeNode*
    If root == NULL
        Return CreateNode(rect)

    If rect.width < root.rect.width
        root.left = Insert(root.left, rect)
    Else
        root.right = Insert(root.right, rect)

    Return root
End Function
```

> 这部分与常规的BST操作相同：
>
> 1. 创建节点时提供矩形结构体作为节点的`rectangle`属性；
> 2. 插入时依据`width`进行排序。

- 寻找最合适的节点

```
Function FindLargestFit(root, targetWidth) -> TreeNode*
    If root == NULL
        Return NULL

    result = NULL

    // If current node's width is greater than targetWidth, search the left
subtree
    If root.rect.width > targetWidth
        result = FindLargestFit(root.left, targetWidth)
    Else
        // If current node fits, search the right subtree
        result = root
        rightResult = FindLargestFit(root.right, targetWidth)
        If rightResult != NULL AND rightResult.rect.width > result.rect.width
            result = rightResult
        End If
    End If

    Return result
End Function
```

> 这个函数是与传统BST查找的较大区别，因为我们需要查找的是**不大于`targetwidth`且宽度最大的矩形。**
>
> 1. 由于我们构建BST时，是利用矩形的`width`属性进行排序，因此我们在搜索时也利用这一属性；
> 2. 如果当前节点的`width`已经大于`targetwidth`，那么直接向其左子树继续搜索；
> 3. 如果当前节点的`width`小于等于`targetwidth`，将返回值`result`先赋值为当前节点`root`，然后继续在其右子树中寻找；
> 4. 递归逻辑如`2,3`步所示，如果找不到符合要求的节点，返回`NULL`。

- **删除指定节点**

```
Function DeleteNode(root, Rectangle rect) -> TreeNode*
    If root == NULL
        Return NULL

    // Search for the node to delete
    If rect.width < root.rect.width
        root.left = DeleteNode(root.left, rect)
    Else If rect.width > root.rect.width
        root.right = DeleteNode(root.right, rect)
    Else
        // Node found

        // Case 1: Leaf node
        If root.left == NULL AND root.right == NULL
            Free root
            Return NULL

        // Case 2: Single child
        If root.left == NULL
            temp = root.right
            Free root
            Return temp

        If root.right == NULL
            temp = root.left
            Free root
            Return temp

        // Case 3: Two children
        // Find the smallest node in the right subtree
        successor = root.right
        successorParent = root

        While successor.left != NULL
            successorParent = successor
            successor = successor.left

        If successorParent != root
            successorParent.left = successor.right
        Else
            successorParent.right = successor.right

        // Copy successor's value to current node
        root.rect = successor.rect
        Free successor
    End If

    Return root
End Function
```

> 1. **递归查找节点**：首先递归查找待删除的节点。如果待删除的矩形宽度小于当前节点的矩形宽度，则向左子树递归；如果大于，则向右子树递归。
> 2. **节点删除的情况**：
>
> - **叶子节点**：如果当前节点没有左子树和右子树，则直接释放该节点并返回 `NULL`。
> - **单个子节点**：如果当前节点只有一个子节点，替换当前节点并返回该子节点。
> - **两个子节点**：如果当前节点有两个子节点，找到右子树中的最小节点（即右子树中最左的节点），用其值替换当前节点，并删除该最小节点。

2.2.2 ceiling_height 相关

- **初始化**

```
void initializeceiling_height(int width) {
    for (int i = 0; i < width; i++) {
        ceiling_height[i] = 0;
    }
}
```

> 初始的容器内部没有矩形，因此数组`ceiling_height`的元素均为`0`。

- **寻找当前数组第一个最低高度的区间**

```
Function FindMaxWidth(containerWidth, startIndex) -> int
    minHeight = FindMinimumHeight(ceiling_height)

    currentStart = -1
    currentWidth = 0

    For each position in the container
```

```
        If position is at the minimum height
            If starting a new segment
                Mark the start
            Increment the width of the current segment
        Else
            If a valid segment was found
                Return the segment's width and start position
            Reset the segment

    If a valid segment reaches the end of the container
        Return the segment's width and start position

End Function
```

> **目的**：我们希望尽可能充分地利用容器，减小最终的高度，因此每次都关注当前`ceiling_height`的最低区域，然后利用`2.2.1`提及的矩形寻找函数，找到最合适的矩形插入（如果找不到，操作请见下文）。
>
> **伪代码解读**：
>
> 1. 首先寻找到当前的最小高度（如果有多个，只需要关注第一个区域，因为相同的最低高度将依次处理，互不影响）
> 2. 然后开始遍历数组，找到第一次出现最低点的连续区域，然后根据情况进行`currentWidth`的自增。
> 3. 如果遍历结束依旧没有返回，说明这一段区域位于数组的末端，需要在`for`循环外直接返回；

- **调整数组**

> 当上述找到的最低高度区间无法插入合适宽度的矩形时，我们需要调整该区域的`ceiling_height`

```
void adjustceiling_height(int startIndex, int width, int containerWidth) {
    // Find the minimum height from the neighboring regions
    // For left side, check the height at startIndex - 1, if it's within bounds
    int leftHeight = (startIndex > 0) ? ceiling_height[startIndex - 1] : INT_MAX;

    // For right side, check the height at startIndex + width, if it's within
bounds
    int rightHeight = (startIndex + width < containerWidth) ?
ceiling_height[startIndex + width] : INT_MAX;

    // The target height is the minimum of the two neighboring heights
    int targetHeight = (leftHeight < rightHeight) ? leftHeight : rightHeight;

    // Now adjust the ceiling_height in the current region
    for (int i = startIndex; i < startIndex + width && i < containerWidth; i++) {
        // If the current section's height is less than the target, adjust it to
the target
        if (ceiling_height[i] < targetHeight) {
            ceiling_height[i] = targetHeight;
        }
    }
}
```

> 配合注释，该部分直接以`c`代码呈现的方式更加明了：
>
> 1. 正如在之前的部分所述，将指定区域的高度调整为相邻区域的较低高度；
> 2. 如果左右某区间不存在，设置为`INT_MAX`无穷大，方便比较取值。

- **更新`ceiling_height`**

> 插入矩形时，更新数组元素

```
void placeRectangle(int startIndex, int rectWidth, int rectHeight) {
    for (int i = startIndex; i < startIndex + rectWidth && i < MAX_WIDTH; i++) {
        ceiling_height[i] += rectHeight;
    }
}
```

2.3 主函数

```
Function CeilingHeightPacking(rectangles, n, containerWidth) -> int
    Initialize the ceiling_height for the container

    totalArea = 0  // Sum of areas of all rectangles
    root = NULL  // Initialize an empty binary search tree (BST)

    For each rectangle in rectangles
        Insert rectangle into the BST
        Add its area to totalArea

    remainingRects = n  // Track remaining rectangles

    While there are remaining rectangles
        // Find the first segment with maximum available width
        maxWidth = FindMaxWidth(containerWidth, startIndex)
        Print "Current maxWidth: maxWidth"

        // Find the best fitting rectangle for the current segment
        bestFit = FindLargestFit(root, maxWidth)

        If bestFit is not NULL
            Print "Placing rectangle with width and height"
            Place the rectangle at the given position
            Remove rectangle from the BST
            Decrease remainingRects
        Else
            Print "No suitable rectangle found. Adjusting ceiling height"
            Adjust the ceiling height in the current region

        Print the updated ceiling_height

    Cleanup the BST
```

```
   caulculate approximation ratio

     Return currentHeight
End Function
```

> 算法的主函数`ceiling_heightPacking`的伪代码如上所示：
>
> 1. `input`：矩形数组、矩形个数、容器宽度；
> 2. `output`：容器最终高度（打印近似比）
> 3. 流程：结合`2.2`的介绍，简要描述：
>   1. 初始化BST以及`ceiling_height`数组，并计算矩形的总面积；
>   2. 当BST当中还存在待插入的矩形时，搜索`ceiling_height`数组，寻找到最低高度的区域然后插入最合适的矩形（视情况调整`ceiling_height`；
>   3. 当BST当中不存在节点时，插入结束。利用容器高度、容器宽度以及矩形的总面积，计算得到近似比并输出，最后返回高度。

Chapter 3: Testing Results

3.1 测试目的及结果

> 输入、输出格式已在`Chapter 1`中说明。

3.1.1 正确性测试

测试目的：验证程序的正确性与鲁棒性

- **样例测试**

`input`:

```
10
5
4 3
3 2
6 5
5 3
7 1
```

`output`:

```
Current maxWidth: 10
Placing rectangle with width: 7, height: 1 at index 0

Current ceiling_height: 1 1 1 1 1 1 1 0 0 0
Current maxWidth: 3
Placing rectangle with width: 3, height: 2 at index 7

Current ceiling_height: 1 1 1 1 1 1 1 2 2 2
Current maxWidth: 7
Placing rectangle with width: 6, height: 5 at index 0

Current ceiling_height: 6 6 6 6 6 6 1 2 2 2
Current maxWidth: 1

No suitable rectangle found. Adjusting ceiling_height at index 6 :
Current ceiling_height: 6 6 6 6 6 6 2 2 2 2
Current maxWidth: 4
Placing rectangle with width: 4, height: 3 at index 6

Current ceiling_height: 6 6 6 6 6 6 5 5 5 5
Current maxWidth: 4

No suitable rectangle found. Adjusting ceiling_height at index 6 :
Current ceiling_height: 6 6 6 6 6 6 6 6 6 6
Current maxWidth: 10
Placing rectangle with width: 5, height: 3 at index 0

Current ceiling_height: 9 9 9 9 9 6 6 6 6 6
```

```
--------------------conclusion--------------------
- Approximation Ratio for ceiling_height packing: 128.5714%
- Final height dealing with 5 rectangles by ceiling_height packing algorithm: 9
```

> 1. 观察过程输出，发现主函数部分正常执行，输出合理。
> 2. 后续将直接展示`conclusion`部分。

- **输入合法性测试**

`input`:

```
5
3
3 4
5 6
7 8
```

`output`:

```
Error: Invalid rectangle size! Width must be > 0 and <= container width, height must be > 0
```

> 输入的矩形宽度大于容器宽度，无法正确插入，程序报错中止。

- **最大输入测试**

`input`: (file: `size_10000.txt`)

```
50
10000
...
```

`output`:

```
--------------------conclusion--------------------
- Approximation Ratio for ceiling_height packing: 100.2564%
- Final height dealing with 10000 rectangles by ceiling_height packing algorithm: 28023
```

3.1.2 高度差异对比测试

测试目的：检查高度范围对近似比的影响

容器宽度固定为`10`，小矩形的宽度范围为`1~10`，不同矩形高度范围下的`Approximation Ratio`：

矩形高度	10	100	1000	10000
较高	113.7500%	104.1260%	102.2160%	100.5645%
较低	115.3846%	105.9524%	102.4718%	100.5248%

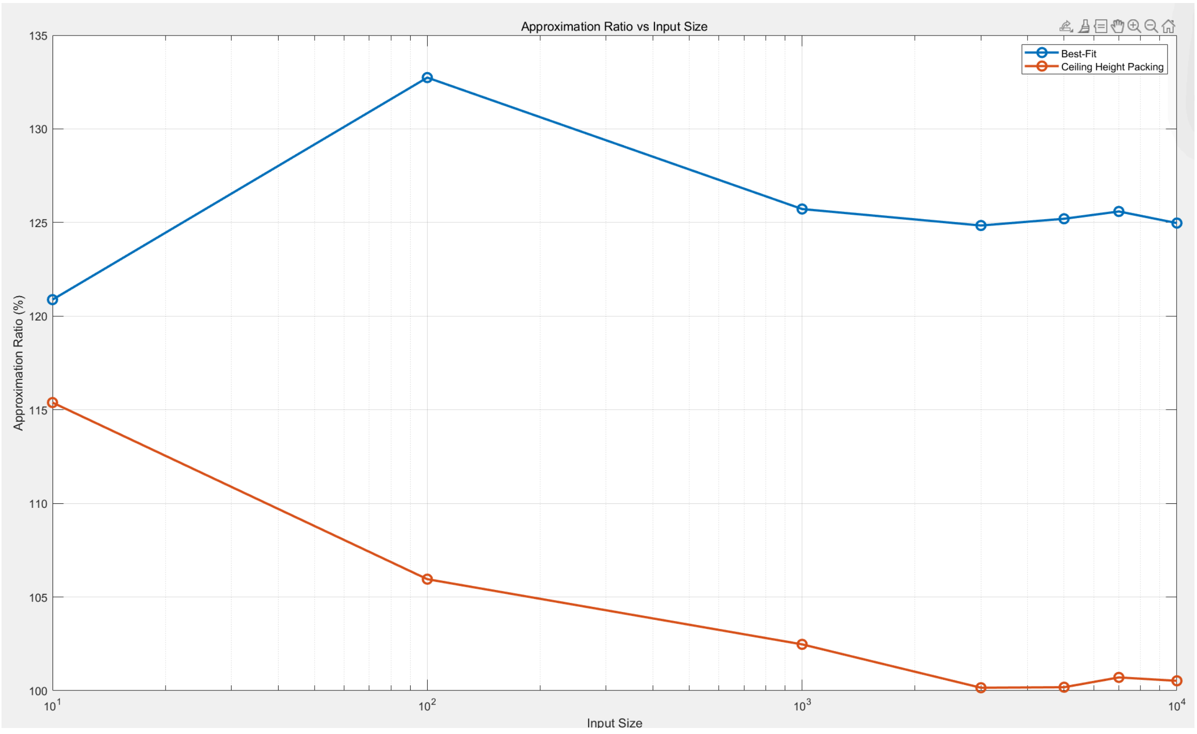
> 1. 较高的范围为`1~50`，较低矩形的高度范围为`1~5`.
> 2. 由实验结果可见，高度差异对近似比的影响较小。但同时当前输入的设置整体有关，从而导致差异不明显。

3.1.3 与`Best Fit`的算法对比

测试目的：与现有的`strip`算法进行比较

用上述的`short`类文件作为输入，不同规模输入下的，二者算法近似比：

算法 / 输入规模	10	100	1000	3000	5000	7000	10000
Best-Fit	120.88%	132.74%	125.72%	124.84%	125.20%	125.59%	124.97%
Ceiling_height packing	115.3846%	105.9524%	102.4718%	100.1554%	100.1825%	100.7064%	100.5248%



1. `best-fit`算法将输入的矩形按照高度排序，优先处理较高的小矩形；每次将对应的矩形插入到最合适的位置（具有最小剩余宽度且能够安放的区间），直至不存在没有插入的矩形；
2. `best-fit`的算法与我们的程序差别在于，前者按照矩形高度进行处理，后者根据当前的容器情况选取最合适宽度的矩形进行处理，具有动态和贪心的特征；
3. 据数学分析，`best-fit` 算法的 **最坏情况近似比率** 约为 **1.7**。即在最坏情况下，算法使用的容器高度最多是理论最优高度的 1.7 倍。

3.2 测试结果分析

1. `3.1.1`的正确性测试表明，对于正常的输入、边界输入以及非法输入，我们的程序都能够分别给出期望的正确输出；
2. `3.1.2`中，我们对于不同规模的输入分别测试了高度较高和较低的矩形输入（对应宽度保持一致），结果的近似比数据显示：高度较高的矩形，其对应结果的近似比往往较低与高度较低的矩形输入组；
3. `3.1.3`，我们对比了我们的算法与`Best-Fit`算法，发现：
    1. 不同规模下，`Best-Fit`返回结果的近似比总是高于我们的算法，且在我们当前的设置（容器宽度为`10`，矩形宽度的分布范围为`1~10`，矩形高度的分布范围为`1~5`)下，近似比趋近于1.25；
    2. `ceiling_height packing`算法的近似比随着输入规模的增大呈现明显的下降趋势，逐渐趋近于1。 这是因为，该算法总是扫描当前的`ceiling_height`数组，然后从BST当中寻找最为合适的矩形。输入规模的扩大意味着每次选取最佳矩形的机会增加，因此越来越接近于最优解。

Chapter 4: Analysis and Comments

4.1 时间复杂度分析

- **BST构建阶段：**

  对n个矩形进行BST插入，每次插入为O(log n),总共需要n次插入操作;

  这一阶段的复杂度为O(`n log n`)

- **装箱循环阶段：**

  while循环执行n次 （n为小矩形的个数，在每次循环中：

```
- findMaxWidth(): O(W)，其中W是容器宽度
- findLargestFit(): O(log n)，在BST中搜索
- deleteNode(): O(log n)，从BST中删除节点
- placeRectangle(): O(W)，更新天际线数组
- adjustSkyline(): O(W)，调整天际线
- 每次循环的复杂度为O(W + log n)
- 总循环复杂度为O(n * (W + log n))
```

因此，**总体时间复杂度**为：O(`n log n + n * (W + log n)`) = O(`n * (W + log n)`)

4.2 空间复杂度分析

- **固定空间：**
  - ceiling_height数组：O(W)，存储天际线高度
  - O(n)，存储矩形数组
- **动态空间：**
  - BST空间：O(n)，存储所有矩形节点
  - 递归调用栈：O(log n)，BST操作的最大递归深度

因此，**总体空间复杂度**为：O(`W + n`)

4.3 算法评价与近似比分析

- **评价**

1. 由时间复杂度分析可知，该算法的时间复杂度与容器宽度`W`相关，当`W`显著大于`n`时，算法的时间复杂度主要与`W`相关。
2. 因此，该算法适用于容器宽度`W`与`n`相比较小的输入，此时算法的近似比随着`n`的输入规模的增大趋近于`1`,具有显著优于`Best-Fit`的表现。
3. 实际中， 算法的近似比率会受到矩形的大小和排列方式的影响。

- **近似比分析**
  - 当小矩形的数量`n`增大时，意味着每次选择最佳矩形进行插入的空间增大，更有利于“完美插入”，此时近似比越低，即 n 与 ρ 成负相关；
  - 记小矩形的宽度 w_i 与容器宽度 W 的比值 w_i/W为 α, 当α减小时，对于相同规模`n`的输入，更有机会寻找到矩形完成”完美插入“，因此此时近似比降低，即 α 与 ρ 成正相关；
  - 当所有的小矩形的宽度都等于 $\frac{1}{2}W_{total} + \mathcal{E}$ 且 $\mathcal{E} > 0$时，每两次搜索中都只能插入一个矩形，空间利用率最小，有近似比的最大值。在此情况下进行**分析**：

$$
\begin{aligned}
W_i &= \frac{1}{2}W_{total} + \mathcal{E} \\
S &= \sum_{i=1}^{n} W_i \cdot h_i = \sum_{i=1}^{n}(\frac{1}{2}W_{total} + \mathcal{E})h_i = H \cdot (\frac{1}{2}W_{total} + \mathcal{E}) \\
H_{\text{opt}} &= \frac{S}{W_{total}} \\
\rho &= \frac{H}{H_{\text{opt}}} = \frac{2 \cdot W_{total}}{W_{total} + 2\mathcal{E}} < 2
\end{aligned}
$$

> 由此可见，该算法在理论上，最坏情况近似比接近于`2`。但是在通常情况下，近似比接近于1，且随输入规模的增大而减小。

Appendix

Source Code

> `solution.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>

// Rectangle structure definition
typedef struct {
    int width;
    int height;
} Rectangle;

// BST node structure
typedef struct TreeNode {
    Rectangle rect;
    struct TreeNode *left;
    struct TreeNode *right;
} TreeNode;

// Define maximum width for ceiling_height
#define MAX_WIDTH 1000
int ceiling_height[MAX_WIDTH];

// BST operations
TreeNode* createNode(Rectangle rect) {
    TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));
    if (node == NULL) {
        fprintf(stderr, "Memory allocation failed!\n");
        exit(1);
    }
    node->rect = rect;
    node->left = node->right = NULL;
    return node;
}

// Insert node (sort by width)
TreeNode* insert(TreeNode* root, Rectangle rect) {
    if (root == NULL) {
        return createNode(rect);
    }

    if (rect.width < root->rect.width) {
        root->left = insert(root->left, rect);
    } else {
        root->right = insert(root->right, rect);
    }
    return root;
}
```

```
TreeNode* findLargestFit(TreeNode* root, int targetWidth){
    if(root == NULL )return NULL;

    TreeNode* result = NULL;

    //if current width is already larger than targetwidth, search the left
subtree
    if(root->rect.width > targetWidth){
        result = findLargestFit(root->left, targetWidth);
    }
    else{
        // If current node fits, search right subtree
        result = root;
        TreeNode* rightResult = findLargestFit(root->right, targetWidth);
        if(rightResult != NULL && rightResult->rect.width > result->rect.width){
            result = rightResult;
        }
    }
    return result;
}

// Delete specified node from BST
TreeNode* deleteNode(TreeNode* root, Rectangle rect) {
    if (root == NULL) return NULL;

    // Search for node to delete
    if (rect.width < root->rect.width) {
        root->left = deleteNode(root->left, rect);
    } else if (rect.width > root->rect.width) {
        root->right = deleteNode(root->right, rect);
    } else {
        // Node found

        // Case 1: Leaf node
        if (root->left == NULL && root->right == NULL) {
            free(root);
            return NULL;
        }

        // Case 2: Single child
        if (root->left == NULL) {
            TreeNode* temp = root->right;
            free(root);
            return temp;
        }
        if (root->right == NULL) {
            TreeNode* temp = root->left;
            free(root);
            return temp;
        }

        // Case 3: Two children
        // Find smallest node in right subtree
        TreeNode* successor = root->right;
        TreeNode* successorParent = root;
        while (successor->left != NULL) {
            successorParent = successor;
            successor = successor->left;
        }

        if (successorParent != root) {
            successorParent->left = successor->right;
        } else {
            successorParent->right = successor->right;
        }

        // Copy successor value to current node
        root->rect = successor->rect;
        free(successor);
    }
    return root;
}

// Clean up entire BST
void cleanupBST(TreeNode* root) {
    if (root != NULL) {
        cleanupBST(root->left);
        cleanupBST(root->right);
        free(root);
    }
}

// ceiling_height related functions
void initializeceiling_height(int width) {
    for (int i = 0; i < width; i++) {
        ceiling_height[i] = 0;
    }
}

void printceiling_height(int containerWidth) {
    printf("Current ceiling_height: ");
    for (int i = 0; i < containerWidth; i++) {
        printf("%d ", ceiling_height[i]);
    }
    printf("\n");
}

// Find the first lowest point in ceiling_height and its width
int findMaxWidth(int containerWidth, int* startIndex) {
    // First find the minimum height
    int minHeight = ceiling_height[0];
    for (int i = 0; i < containerWidth; i++) {
        if (ceiling_height[i] < minHeight) {
            minHeight = ceiling_height[i];
        }
    }

    // Then find the first wide area at the minimum height
    int currentStart = -1;
    int currentWidth = 0;
    for (int i = 0; i < containerWidth; i++) {
        if (ceiling_height[i] == minHeight) {
            if (currentWidth == 0) {
                currentStart = i;  // Mark the start of this segment
            }
            currentWidth++;
        } else {
            if (currentWidth > 0) {
                // If we encountered a segment of minimum height, return it
                *startIndex = currentStart;
                return currentWidth;
            }
            currentWidth = 0;  // Reset for the next possible segment
        }
    }

    // Handle the case where the segment ends at the last position
    if (currentWidth > 0) {
        *startIndex = currentStart;
        return currentWidth;
    }

    return 0;  // Return 0 if no valid segment found (shouldn't happen under
normal conditions)
}


void placeRectangle(int startIndex, int rectWidth, int rectHeight) {
    for (int i = startIndex; i < startIndex + rectWidth && i < MAX_WIDTH; i++) {
        ceiling_height[i] += rectHeight;
    }
}

int calculateHeight(int containerWidth) {
    int maxHeight = 0;
    for (int i = 0; i < containerWidth; i++) {
        if (ceiling_height[i] > maxHeight) {
            maxHeight = ceiling_height[i];
        }
    }
    return maxHeight;
}

// Improved ceiling_height adjustment strategy
void adjustceiling_height(int startIndex, int width, int containerWidth) {
    // Find the minimum height from the neighboring regions
    // For left side, check the height at startIndex - 1, if it's within bounds
    int leftHeight = (startIndex > 0) ? ceiling_height[startIndex - 1] : INT_MAX;

    // For right side, check the height at startIndex + width, if it's within
bounds
    int rightHeight = (startIndex + width < containerWidth) ?
ceiling_height[startIndex + width] : INT_MAX;

    // The target height is the minimum of the two neighboring heights
    int targetHeight = (leftHeight < rightHeight) ? leftHeight : rightHeight;
    // Now adjust the ceiling_height in the current region
    for (int i = startIndex; i < startIndex + width && i < containerWidth; i++) {
        // If the current section's height is less than the target, adjust it to
the target
        if (ceiling_height[i] < targetHeight) {
            ceiling_height[i] = targetHeight;
        }
    }
}



// Main algorithm: ceiling_height Packing using BST
int ceiling_heightPacking(Rectangle *rectangles, int n, int containerWidth) {
    initializeceiling_height(containerWidth);

    int area_sum = 0; //calculate the sum of all areas
    // Build BST
    TreeNode* root = NULL;
    for (int i = 0; i < n; i++) {
        root = insert(root, rectangles[i]);
        area_sum += rectangles[i].width * rectangles[i].height;
    }

    int remainingRects = n;

    while (remainingRects > 0) {
        int startIndex = 0;
        int maxWidth = findMaxWidth(containerWidth, &startIndex);
        printf("Current maxWidth: %d\n", maxWidth);

        // Find suitable rectangle
        TreeNode* bestFit = findLargestFit(root, maxWidth);

        if (bestFit != NULL) {
            Rectangle bestRect = bestFit->rect;
            printf("Placing rectangle with width: %d, height: %d at index
%d\n\n",
                   bestRect.width, bestRect.height, startIndex);

            // Place rectangle
            placeRectangle(startIndex, bestRect.width, bestRect.height);

            // Remove used rectangle from BST
            root = deleteNode(root, bestRect);

            remainingRects--;
        } else {
            printf("\nNo suitable rectangle found. Adjusting ceiling_height at
index %d :\n", startIndex);
            adjustceiling_height(startIndex, maxWidth, containerWidth);
        }

        printceiling_height(containerWidth);
    }
```

```c
    // Clean up BST
    cleanupBST(root);

    if (remainingRects > 0) {
        printf("Warning: Could not place all rectangles efficiently\n");
    }

    float optimal_height = area_sum / (float)containerWidth;
    int  current_height = calculateHeight(containerWidth);
    float radio = current_height / optimal_height * 100;
    // printf("The sum of all areas is: %d\n", area_sum);

    printf("\n--------------------conclusion---------------------\n");
    printf("- Approximation Ratio for ceiling_height packing: %.4f%%\n", radio);

    return current_height;
}
```

`correctness_test.c`

```c
#include "solution.h"
#include <stdio.h>
#include <stdlib.h>

// 函数声明
TreeNode* createNode(Rectangle rect);
TreeNode* insert(TreeNode* root, Rectangle rect);
TreeNode* findLargestFit(TreeNode* root, int targetWidth);
TreeNode* deleteNode(TreeNode* root, Rectangle rect);
void cleanupBST(TreeNode* root);
void initializeceiling_height(int width);
void printceiling_height(int containerWidth);
int findMaxWidth(int containerWidth, int* startIndex);
void placeRectangle(int startIndex, int rectWidth, int rectHeight);
int calculateHeight(int containerWidth);
void adjustceiling_height(int startIndex, int width, int containerWidth);
int ceiling_heightPacking(Rectangle *rectangles, int n, int containerWidth);

// 全局变量
int ceiling_height[MAX_WIDTH];

int main() {
    int containerWidth = 10; //max width of the container

    //initialize rectangles
    Rectangle rectangles[] = {
        {4, 3}, {3, 2}, {6, 5}, {5, 3}, {7, 1}
    };
    int n = sizeof(rectangles) / sizeof(rectangles[0]);

    int finalHeight = ceiling_heightPacking(rectangles, n, containerWidth);
//call the function to pack the rectangles
    printf("\nFinal container height: %d\n", finalHeight);
    return 0;
}
```

`file_input.c`

```
#include "solution.h"
#include <stdio.h>
#include <stdlib.h>

// Function declarations
TreeNode* createNode(Rectangle rect);
TreeNode* insert(TreeNode* root, Rectangle rect);
TreeNode* findLargestFit(TreeNode* root, int targetWidth);
TreeNode* deleteNode(TreeNode* root, Rectangle rect);
void cleanupBST(TreeNode* root);
void initializeceiling_height(int width);
void printceiling_height(int containerWidth);
int findMaxWidth(int containerWidth, int* startIndex);
void placeRectangle(int startIndex, int rectWidth, int rectHeight);
int calculateHeight(int containerWidth);
void adjustceiling_height(int startIndex, int width, int containerWidth);
int ceiling_heightPacking(Rectangle *rectangles, int n, int containerWidth);

// Global variable
int ceiling_height[MAX_WIDTH];

int main() {
    FILE *fp;
    int containerWidth; // Maximum width of the container
    int n; // Number of rectangles

    fp = fopen("Generate_input/size_7000_short.txt", "r"); // Open input file,
please change the file name to your own
    if (fp == NULL) {
        printf("Error: Cannot open input file\n");
        return 1;
    }

    fscanf(fp, "%d", &containerWidth);
    fscanf(fp, "%d", &n);

    Rectangle* rectangles = (Rectangle*)malloc(n * sizeof(Rectangle));
    if (rectangles == NULL) {
        fprintf(stderr, "Memory allocation failed!\n");
        fclose(fp);
        return 1;
    }

    for (int i = 0; i < n; i++) {
        fscanf(fp, "%d %d", &rectangles[i].width, &rectangles[i].height);

        //check if the rectangle size is valid
        if (rectangles[i].width <= 0 || rectangles[i].height <= 0 ||
            rectangles[i].width > containerWidth) {
```

```
            printf("Error: Invalid rectangle size! Width must be > 0 and <=
container width, height must be > 0\n");
            free(rectangles);
            fclose(fp);
            return 1;
        }
    }

    fclose(fp);

    int finalHeight = ceiling_heightPacking(rectangles, n, containerWidth);  //
Call the ceiling_height packing function

    printf("- Final height dealing with %d rectangles by ceiling_height packing
algorithm: %d\n", n, finalHeight);

    free(rectangles);
    return 0;
}
```

`manual_input.c`

```
#include "solution.h"
#include <stdio.h>
#include <stdlib.h>

// Function declarations
TreeNode* createNode(Rectangle rect);
TreeNode* insert(TreeNode* root, Rectangle rect);
TreeNode* findLargestFit(TreeNode* root, int targetWidth);
TreeNode* deleteNode(TreeNode* root, Rectangle rect);
void cleanupBST(TreeNode* root);
void initializeceiling_height(int width);
void printceiling_height(int containerWidth);
int findMaxWidth(int containerWidth, int* startIndex);
void placeRectangle(int startIndex, int rectWidth, int rectHeight);
int calculateHeight(int containerWidth);
void adjustceiling_height(int startIndex, int width, int containerWidth);
int ceiling_heightPacking(Rectangle *rectangles, int n, int containerWidth);

// Global variable
int ceiling_height[MAX_WIDTH];

int main() {
    int containerWidth; //maximum width of the container
    int n; //number of rectangles

    printf("Enter container width: ");
    scanf("%d", &containerWidth);

    printf("Enter number of rectangles: ");
    scanf("%d", &n);

    Rectangle* rectangles = (Rectangle*)malloc(n * sizeof(Rectangle));
    if (rectangles == NULL) {
        fprintf(stderr, "Memory allocation failed!\n");
        return 1;
    }

    // Read input rectangles
    for (int i = 0; i < n; i++) {
        printf("Enter width and height for rectangle %d (format: width height):
", i + 1);
        scanf("%d %d", &rectangles[i].width, &rectangles[i].height);

        // Check for invalid input
        if (rectangles[i].width <= 0 || rectangles[i].height <= 0 ||
            rectangles[i].width > containerWidth) {
            printf("Error: Invalid rectangle size! Width must be > 0 and <=
container width, height must be > 0\n");
            free(rectangles);
            return 1;
        }
    }

    int finalHeight = ceiling_heightPacking(rectangles, n, containerWidth); //
Pack rectangles into container
    printf("\nFinal container height: %d\n", finalHeight);

    free(rectangles); // Free memory
    return 0;
}
```

`Generator.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Define the maximum container width and number of rectangles
#define CONTAINER_WIDTH 10
#define NUMBER 7000

// Function to generate random integers within range [min, max]
int random_int(int min, int max) {
    return min + rand() % (max - min + 1);
}

int main() {
    // Set random seed
    srand((unsigned int)time(NULL));

    // Open the output files for short and high
    FILE *fileShort = fopen("size_7000_short.txt", "w");
    FILE *fileHigh = fopen("size_7000_high.txt", "w");

    if (fileShort == NULL || fileHigh == NULL) {
        printf("Failed to create one or both of the files!\n");
        return 1;
    }
```

```c
    // Write container_width and number to both files
    fprintf(fileShort, "%d\n", CONTAINER_WIDTH);
    fprintf(fileShort, "%d\n", NUMBER);

    fprintf(fileHigh, "%d\n", CONTAINER_WIDTH);
    fprintf(fileHigh, "%d\n", NUMBER);

    // Generate rectangle data for both files
    for (int i = 0; i < NUMBER; i++) {
        int width = random_int(1, CONTAINER_WIDTH);  // Ensure width does not
exceed CONTAINER_WIDTH
        int shortHeight = random_int(1, 5);          // Short height is within
range [1, 5]

        int highHeight = random_int(1, 50);

        // Write data to the short file
        fprintf(fileShort, "%d %d\n", width, shortHeight);

        // Write data to the high file (height is independently random within
[shortHeight*10, shortHeight*10 + 9])
        fprintf(fileHigh, "%d %d\n", width, highHeight);
    }

    // Close the files
    fclose(fileShort);
    fclose(fileHigh);

    printf("Data has been generated for both short and high files\n");
    return 0;
}
```

> `best_fit.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>

#define MAX_WIDTH 1000

typedef struct {
    int width;
    int height;
} Rectangle;

typedef struct {
    int x;      // x coordinate
    int height; // current height at this x coordinate
} Position;

// Compare rectangles by height in descending order
int compareRectangles(const void *a, const void *b) {
    Rectangle *rectA = (Rectangle *)a;
    Rectangle *rectB = (Rectangle *)b;
    return rectB->height - rectA->height;
}

// Find the position with minimum height that can fit the current rectangle
int findBestFit(Position *positions, int containerWidth, int rectWidth, int
*minX) {
    int minHeight = INT_MAX;
    int bestX = -1;

    // Try each possible x position
    for (int x = 0; x <= containerWidth - rectWidth; x++) {
        // Find maximum height in the range [x, x+width-1]
        int maxHeight = 0;
        bool canFit = true;

        for (int i = x; i < x + rectWidth; i++) {
            if (positions[i].height > maxHeight) {
                maxHeight = positions[i].height;
            }
        }

        // If this position has the lowest height so far, update best position
        if (maxHeight < minHeight) {
            minHeight = maxHeight;
            bestX = x;
        }
    }

    *minX = bestX;
    return minHeight;
}

int bestFitPacking(Rectangle *rectangles, int n, int containerWidth) {
    qsort(rectangles, n, sizeof(Rectangle), compareRectangles);

    // Initialize positions array to track height at each x coordinate
    Position *positions = (Position *)calloc(containerWidth, sizeof(Position));
    for (int i = 0; i < containerWidth; i++) {
        positions[i].x = i;
        positions[i].height = 0;
    }

    float area_sum = 0;
    int totalHeight = 0;

    // Place each rectangle
    for (int i = 0; i < n; i++) {
        Rectangle currentRect = rectangles[i];
        int bestX;
        int baseHeight = findBestFit(positions, containerWidth,
currentRect.width, &bestX);

        if (bestX != -1) {
            // Update heights for the width of the rectangle
            int newHeight = baseHeight + currentRect.height;
            for (int x = bestX; x < bestX + currentRect.width; x++) {
                positions[x].height = newHeight;
            }

            // Update total height if necessary
            if (newHeight > totalHeight) {
                totalHeight = newHeight;
            }
        }

        area_sum += currentRect.width * currentRect.height;
    }

    // Calculate and output metrics
    float optimal_height = area_sum / containerWidth;
    float ratio = totalHeight / optimal_height * 100;

    printf("\nOptimal height (area/width): %.2f\n", optimal_height);
    printf("Actual container height: %d\n", totalHeight);
    printf("Total rectangle area: %.2f\n", area_sum);
    printf("Approximation Ratio for Best Fit packing: %.2f%%\n", ratio);
    printf("\n--------------------conclusion---------------------\n");


    free(positions);
    return totalHeight;
}

int main() {
    FILE *fp;
    int containerWidth; // Maximum width of the container
    int n; // Number of rectangles

    fp = fopen("../Generate_input/size_7000_short.txt", "r"); // Open input file,
please change the file name to your own
    if (fp == NULL) {
        printf("Error: Cannot open input file\n");
        return 1;
    }

    fscanf(fp, "%d", &containerWidth);
    fscanf(fp, "%d", &n);

    Rectangle* rectangles = (Rectangle*)malloc(n * sizeof(Rectangle));
    if (rectangles == NULL) {
        fprintf(stderr, "Memory allocation failed!\n");
        fclose(fp);
        return 1;
    }

    for (int i = 0; i < n; i++) {
        fscanf(fp, "%d %d", &rectangles[i].width, &rectangles[i].height);
        // Check if the rectangle size is valid
        if (rectangles[i].width <= 0 || rectangles[i].height <= 0 ||
            rectangles[i].width > containerWidth) {
```

```
            printf("Error: Invalid rectangle size! Width must be > 0 and <=
container width, height must be > 0\n");
            free(rectangles);
            fclose(fp);
            return 1;
        }
    }

    fclose(fp);

    // Call the Best Fit packing function

    int finalHeight = bestFitPacking(rectangles, n, containerWidth);
    printf("\nFinal container height with Best Fit packing: %d\n", finalHeight);

    // Free the allocated memory for rectangles
    free(rectangles);
    return 0;
}
```

> 具体的关系请参见`README`

References

> None.

Author List

--

Declaration

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