Texture Packing

Chapter 1: Introduction

1.1 Background

Texture packing is a critical problem in the field of computer graphics, where the goal is to efficiently pack multiple rectangle-shaped textures into a single large texture. This process is essential in various applications, including video game development, 3D rendering, and image processing, where minimizing the texture memory usage and optimizing rendering performance are crucial.

The primary objective of texture packing is to arrange the given textures within a large texture of specified width while minimizing the height required. This is akin to a two-dimensional bin packing problem, which is known to be NP-hard. Hence, exact solutions are computationally infeasible for large instances, making approximation algorithms a practical choice.

In this project, the focus is on designing an approximation algorithm that runs in polynomial time. The algorithm must be capable of handling a wide range of input sizes, from small test cases of 10 textures to large cases with up to 10,000 textures. Additionally, the textures will have varying distributions of widths and heights, which introduces further complexity to the problem.

To thoroughly evaluate the performance of the proposed algorithm, it is essential to generate diverse test cases and analyze the factors influencing the approximation ratio. Factors such as the distribution of texture dimensions, the ratio of width to height, and the overall size of the texture set are expected to impact the effectiveness of the algorithm. The analysis aims to identify these factors and understand their implications on the algorithm's performance, ensuring a robust and efficient solution for texture packing in practical applications.

1.2 Project Overview

The texture packing project aims to develop an efficient approximation algorithm to pack multiple rectangle-shaped textures into a single large texture with a given width and a minimized height. The project entails several critical components and steps to ensure the algorithm's robustness and practicality in real-world applications.

Objectives

1. **Designing an Approximation Algorithm**: The core of this project is to create an approximation algorithm that operates in polynomial time. The algorithm should be capable of efficiently arranging the given textures within the constraints of the specified width and minimized height.
2. **Test Case Generation**: To evaluate the performance of the proposed algorithm, we use Python to generate various test cases to evaluate the performance of the proposed approximation algorithm. The provided code includes functions to create different types of test cases, ensuring a comprehensive assessment of the algorithm's capabilities under diverse scenarios.
3. **Performance Analysis**: A thorough analysis of the algorithm's performance will be conducted, focusing on factors that influence the approximation ratio. These factors may include the distribution of texture dimensions, the width-to-height ratio, and the overall size of the texture set. Understanding these influences will help in refining the algorithm and ensuring its effectiveness.

Methodology

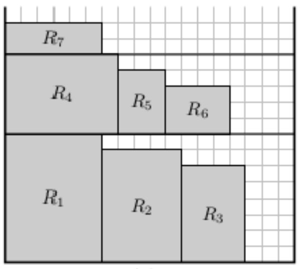
We employs multiple algorithms to tackle the texture packing problem, where the objective is to pack a given number of rectangles into a large bin of fixed width while minimizing the height. Each algorithm utilizes a distinct approach to achieve this goal. Here is a detailed explanation of the problem-solving strategies for each algorithm:

1. Next Fit Decreasing Height (NFDH)

Solution Approach:

Input Rectangles $\mathcal{I}$

i	1	2	3	4	5	6	7
$w(i)$	6	5	4	7	3	4	6
$h(i)$	8	7	6	5	4	3	2



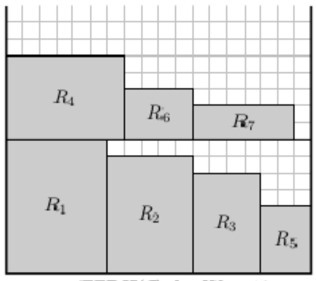
NFDH($\mathcal{I}$) for $W = 18$

- **Sorting**: The rectangles are first sorted in decreasing order of height. This ensures that taller rectangles are placed first, which helps in optimizing space usage vertically.
- **Packing Strategy**: The algorithm places each rectangle into the current bin (row) if it fits. If the rectangle does not fit in the current bin, a new bin is created, and the rectangle is placed in this new bin.
- **Advantages**: This approach reduces the complexity of checking multiple bins for space, as it only considers the current bin, making it faster but potentially less space-efficient than FFDH.

2. First Fit Decreasing Height (FFDH)

Input Rectangles $\mathcal{I}$

i	1	2	3	4	5	6	7
$w(i)$	6	5	4	7	3	4	6
$h(i)$	8	7	6	5	4	3	2



FFDH($\mathcal{I}$) for $W = 18$

Solution Approach:

- **Sorting**: Similar to NFDH, the rectangles are sorted in decreasing order of height.
- **Packing Strategy**: The algorithm iterates over the sorted list of rectangles, placing each rectangle into the first bin (row) that has enough remaining width to accommodate it. If no such bin exists, a new bin is created.
- **Advantages**: This method is straightforward and efficient for minimizing the height of the packing by ensuring that the tallest rectangles are placed first, reducing the chance of creating excessively high bins later on.

3. Sleator

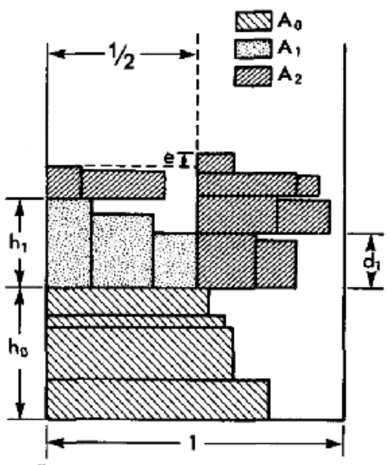


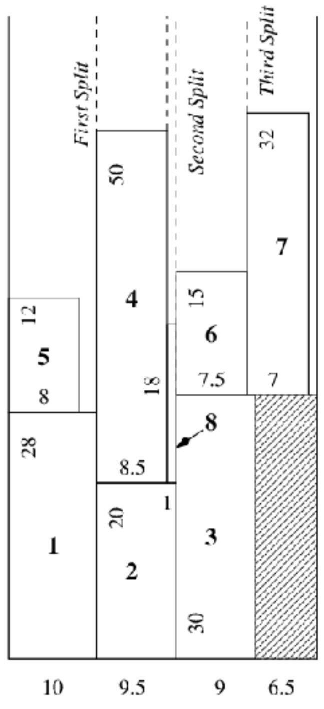
Fig. 1. The anatomy of a packing.

Solution Approach:

- **Partitioning**: Rectangles are divided into two groups based on their width relative to the given strip width: Group A (width > half of the strip width) and Group B (width <= half of the strip width).
- **Packing Strategy**:
  - **Group A**: Rectangles are placed vertically along the left side of the strip, one above the other.
  - **Group B**: Rectangles are placed horizontally at the bottom or top of the strip, balancing the height on both ends.
- **Advantages**: This method effectively utilizes the width of the strip and balances the height by strategically placing rectangles based on their widths.

4. Split Packing (SP)

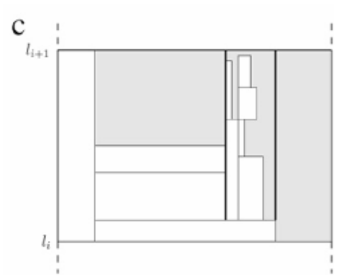
Solution Approach:



- **Sorting**: Rectangles are sorted by width and height to prioritize placing larger rectangles first.
- **Packing Strategy**: The algorithm creates strips to accommodate rectangles, cutting strips into sub-strips as needed to fit the rectangles. This method allows for efficient use of space by creating flexible packing regions within the large bin.

- **Advantages**: Split Packing can handle a diverse set of rectangle sizes and efficiently use the available space by dynamically adjusting strip sizes.

5. Size Alternating Stack (SAS)



Solution Approach:

- **Partitioning**: Rectangles are divided into two categories: narrow (width < height) and wide (width >= height).
- **Sorting**: Narrow rectangles are sorted by height, and wide rectangles are sorted by width. If the width (or height) equals,then sort them by height(or width). This prioritization helps in organizing the packing more efficiently.
- **Packing Strategy**: The packing process alternates between placing narrow and wide rectangles, starting with the taller rectangles first. This alternation helps in balancing the use of space and minimizing the height incrementally.
- **Advantages**: By categorizing and sorting rectangles, SAS efficiently uses space and reduces the overall height required for packing, which performs well when the maximum width of the rectangles is close to the width of the container.

6. Advanced Size Alternating Stack (ad_SAS)

However, when the container width is much larger than the width of the rectangles, SAS algorithm performs poorly. Therefore, we optimized this algorithm and designed ad_SAS algorithm.

Solution Approach:

- **Adaptive Partitioning**: Similar to SAS, rectangles are divided into two categories: narrow (width < height) and wide (width >= height).
- **Sorting**: Narrow rectangles are sorted by height, and wide rectangles are sorted by width.
- **Packing Strategy**: When there is surplus space in the horizontal direction, this algorithm continues to pack wide until the horizontal space is maximally utilized.
- **Advantages**: This algorithm effectively addresses the issue of SA's inefficient utilization in the horizontal direction.

Expected Outcomes

1. **Efficient Algorithm**: A polynomial-time approximation algorithm that effectively packs textures with a minimized height, suitable for practical applications in computer graphics.
2. **Comprehensive Test Suite**: A diverse set of test cases that can be used for further research and development in texture packing algorithms.
3. **Insightful Analysis**: Detailed analysis of factors affecting the algorithm's performance, providing valuable insights for future improvements and applications.

By the end of this project, the developed algorithm and the accompanying analysis are expected to contribute significantly to the field of texture packing, offering a practical solution that can be applied in various graphics-related applications.

Chapter2: Algorithm Specification

2.1 Data Structure and Variables by Algorithm

We implements several algorithms for the texture packing problem. Each algorithm utilizes specific data structures and variables to achieve its functionality. Here is a summary categorized by each algorithm:

1. First Fit Decreasing Height (FFDH)

- **Data Structures**:
  - `vector<Rectangle> rects` : List of rectangles to be packed.
- **Functions**:
  - `bool cmpBins(const Rectangle& a, const Rectangle& b)` : Comparator function to sort rectangles by height.
  - `double FFDH(vector<Rectangle>& rects)` : Function to implement the FFDH algorithm.

2. Next Fit Decreasing Height (NFDH)

- **Data Structures**:
  - `vector<Rectangle> rects` : List of rectangles to be packed.
- **Functions**:
  - `bool cmpBins(const Rectangle& a, const Rectangle& b)` : Comparator function to sort rectangles by height.
  - `double NFDH(vector<Rectangle>& rects)` : Function to implement the NFDH algorithm.

3. Size Alternating Stack(SAS)

- **Data Structures**:
  - `vector<Rectangle> narrow` : List of narrow rectangles (width < height).
  - `vector<Rectangle> wide` : List of wide rectangles (width >= height).
- **Functions**:
  - `bool cmpHeight(const Rectangle& a, const Rectangle& b)` : Comparator function to sort rectangles by height.
  - `bool cmpWide(const Rectangle& a, const Rectangle& b)` : Comparator function to sort rectangles by width.
  - `double SAS(vector<Rectangle>& rects)` : Function to implement the SAS algorithm.
  - `void PackNarrow(vector<Rectangle>& narrow, vector<Rectangle>& wide, double x1, double y1, double x_limit, double y_limit)` : Helper function to pack narrow rectangles.
  - `void PackWide(vector<Rectangle>& narrow, vector<Rectangle>& wide, double x1, double y1, double x_limit, double y_limit)` : Helper function to pack wide rectangles.
- **Pseudocode**:

```
Define PackNarrow(narrow, wide, x1, y1, x_limit, y_limit):
  While narrow is not empty and fits within x_limit:
    Set baseWidth to current narrow width
    While narrow fits within y_limit and baseWidth:
      Pack narrow rectangle at current coordinates
      Remove narrow rectangle
    Update current_x and current_y
Define PackWide(narrow, wide, x1, y1, x_limit, y_limit):
  While wide is not empty and fits within x_limit:
    For each wide rectangle:
            If it fits within y_limit:
                If remaining width can fit narrow, pack narrow
                Pack wide rectangle at current coordinates
                Remove wide rectangle
        If no more wide rectangles but narrow rectangles remain, pack narrow
```

4. Advanced Size Alternating Stack (ad_SAS)

- **Data Structures**:
  - `vector<Rectangle> narrow` : List of narrow rectangles (width < height).
  - `vector<Rectangle> wide` : List of wide rectangles (width >= height).
- **Functions**:
  - `bool cmpHeight(const Rectangle& a, const Rectangle& b)` : Comparator function to sort rectangles by height.
  - `bool cmpWide(const Rectangle& a, const Rectangle& b)` : Comparator function to sort rectangles by width.
  - `double ad_SAS(vector<Rectangle>& rects)` : Function to implement the adaptive SAS algorithm.
  - `void ad_PackNarrow(vector<Rectangle>& narrow, vector<Rectangle>& wide, double x1, double y1, double x_limit, double y_limit)` : Advanced helper function to pack narrow rectangles.
  - `void ad_PackWide(vector<Rectangle>& narrow, vector<Rectangle>& wide, double x1, double y1, double x_limit, double y_limit)` : Advanced helper function to pack wide rectangles.
- **Pseudocode**:

```
Function ad_PackNarrow(narrow, wide, x1, y1, x_limit, y_limit):
    While narrow is not empty and fits within x_limit:
        Set baseWidth to current narrow width
        While narrow fits within y_limit and baseWidth:
            Pack narrow rectangle at (curr_X1, curr_Y1)
            Update curr_Y1
            Remove packed narrow rectangle
        Update curr_X1
        Reset curr_Y1 to y1
    If narrow is empty and wide can fit:
        Call ad_PackWide with updated parameters

Function ad_PackWide(narrow, wide, x1, y1, x_limit, y_limit):
    While wide is not empty and fits within x_limit:
        For each wide rectangle:
            If wide fits within y_limit:
                If first wide rectangle:
                    Update first_width
                If remaining width can fit narrow:
                    Call ad_PackNarrow with updated parameters
                Pack wide rectangle at (x1, y1)
                Update x_limit and y1
                Remove packed wide rectangle
    If right space can fit wide:
        Call ad_PackWide with updated parameters
        Return
    If wide is empty but narrow can fit:
        Call ad_PackNarrow with updated parameters
```

5. Sleator

- **Data Structures**:
  - `vector<Rectangle> groupA` : List of rectangles with width greater than half of the strip width.
  - `vector<Rectangle> groupB` : List of rectangles with width less than or equal to half of the strip width.
- **Functions**:
  - `double Sleator(vector<Rectangle>& rects)` : Function to implement the Sleator algorithm.
- **Pseudocode**:

```
Divide rectangles into Group A (width > half strip width) and Group B (width <= half strip width)
Initialize currentHeight to 0
Place items from Group A vertically, updating currentHeight
Sort Group B by height in descending order
Initialize lowerHeight and upperHeight to currentHeight
Place items from Group B horizontally, balancing between lowerHeight and upperHeight:
    For each item in Group B:
        Determine maximum height that can be achieved by placing items horizontally
        Update either lowerHeight or upperHeight
Return the maximum of lowerHeight and upperHeight
```

6. Split Packing (SP)

- **Data Structures**:
  - `vector<Rectangle> rects` : List of rectangles to be packed.
  - `vector<Strip> strips` : List of strips used for packing.
- **Functions**:
  - `bool cmp(const Rectangle& a, const Rectangle& b)` : Comparator function to sort rectangles by width and height.
  - `double SP(vector<Rectangle>& rects)` : Function to implement the Split Packing algorithm.
- **Pseudocode**:

```
for each rectangle i in recs:
    s_index = 0  // Initialize strip index to 0
    // Find the first strip that can accommodate the rectangle
    for each strip j from 1 to strip_num:
        if strip j can fit rectangle i:
            if no strip selected yet:
                select strip j
            else if strip j is narrower than currently selected strip:
                select strip j
    if no suitable strip found:
        // Find the strip with the lowest upper bound
        for each strip j from 1 to strip_num:
            if no strip selected yet or strip j has a lower upper bound:
                select strip j
        update selected strip's lower and upper bounds
        set strip's item width to rectangle's width
    else:
        // Suitable strip found, place the item and create two substrips
        create s1 using selected strip's item width and upper bound
        create s2 using rectangle's width and selected strip's lower bound

        remove selected strip from list
        insert s1 and s2 into the list
        increment strip_num
```

Utility Functions

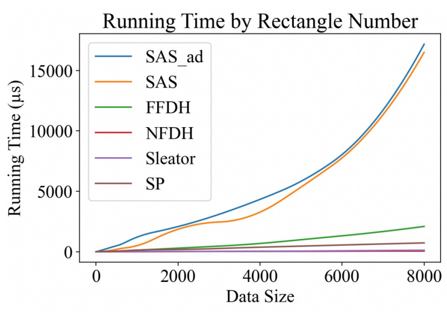
- **Comparator Functions**:
    - `bool cmp(const Rectangle& a, const Rectangle& b)` : Comparator function to sort rectangles by width and then by height or vice versa.
- **Timing Function**:
    - `void testTime(vector<Rectangle>& rects)` : Function to test the execution time of different algorithms.

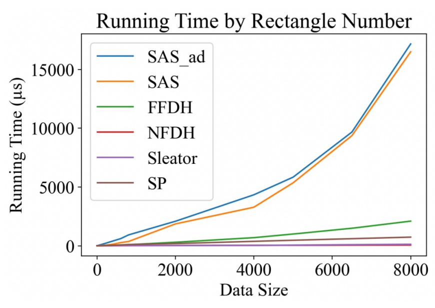
Chapter3: Testing Results

We specifically wrote a Python script to generate test cases and test programs, and we conducted testing and charting of the program as needed.

Running Time

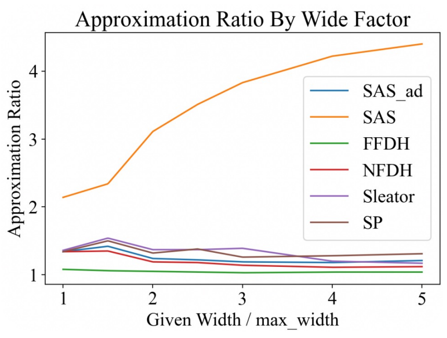
We analyzed the runtime of each algorithm under different data scales individually and organized the results into the following chart.



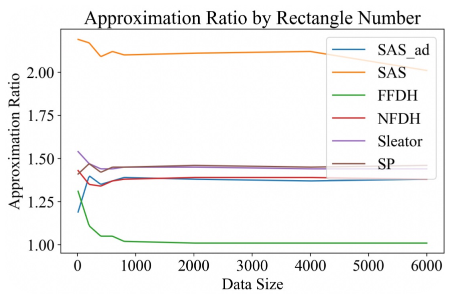


Approximation ratio

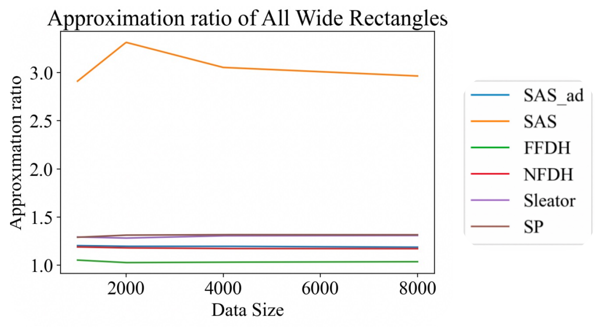
We tested and evaluated each algorithm under random conditions and extreme cases, including scenarios with all wide rectangles, all narrow rectangles, all squares, and so on. The performance of each algorithm varies in different situations, which perhaps is the charm of approximation algorithms.

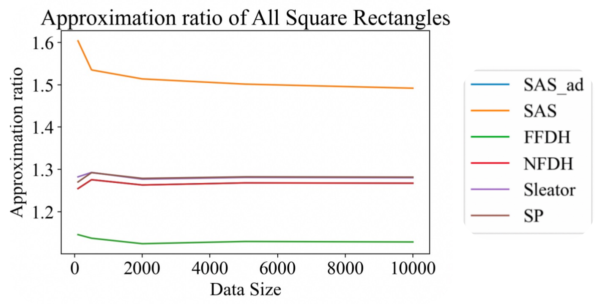


The FFDH algorithms perform the best, maintaining low and less fluctuating approximation ratios. The NFDH and SAS_advanced performs good and stable. Sleator and SP are Stable, yet a little inferior to those above. SAS, no wonder performs poorly with increasing given width.



The FFDH algorithms perform the best, maintaining low and less fluctuating approximation ratios. The NFDH and SAS_advanced performs good and stable. Sleator and SP are Stable, yet a little inferior to those above. SAS, no wonder performs poorly in common cases.





When all the rectangles are in the same shape, **SAS_advanced is very close to NFDH**, simply stacking one rectangle after another.
When it comes to All **Narrow** Rectangles, SP performs poorly,
wasting a lot spaces, we suppose it comes from the order of decreasing width instead of height.

Chapter4: Analysis and Comments

Here is the detailed analysis of the time and space complexity for each of the implemented algorithms:

1. First Fit Decreasing Height (FFDH)

- **Time Complexity**:
  - **Sorting**: Sorting the rectangles by height takes $O(n \log n)$.
  - **Packing**: In the worst case, each rectangle might be compared with all previous bins, resulting in $O(n^2)$ comparisons.
  - **Overall**: $O(n \log n) + O(n^2) = O(n^2)$.
- **Space Complexity**:
  - **Storage for Rectangles**: $O(n)$ to store the list of rectangles.
  - **Current Widths**: $O(n)$ for the array tracking the current width of each bin.
  - **Overall**: $O(n)$.
- **Approximation ratio**:
  - $FFDH(I) \leq 1.7OPT(I) + hmax \leq 2.7OPT(I)$

2. Next Fit Decreasing Height (NFDH)

- **Time Complexity**:
  - **Sorting**: Sorting the rectangles by height takes $O(n \log n)$.
  - **Packing**: Each rectangle is only compared with the current bin, so packing takes $O(n)$.
  - **Overall**: $O(n \log n) + O(n) = O(n \log n)$.
- **Space Complexity**:
  - **Storage for Rectangles**: $O(n)$.
  - **Current Widths**: $O(n)$ for the array tracking the current width of each bin.
  - **Overall**: $O(n)$.
- **Approximation ratio**:

$$
\begin{aligned}
&\text{NFDH}(I) = \sum_{\text{i=1}}^{t} H_i \leq H_1 + \frac{\sum_{i=1}^{t-1} S_i + \sum_{i=2}^{t} S_i}{W} \\
&\leq H_1 + 2 \times \frac{S}{W} \\
&\leq h_{\max} + 2OPT(I) \\
&\leq 3OPT(I)
\end{aligned}
$$

3. Size Alternating Stack (SAS)

- **Time Complexity**:
  - **Partitioning**: Dividing rectangles into narrow and wide categories takes $O(n)$.
  - **Sorting**: Sorting narrow and wide rectangles takes $O(n \log n)$ each, totaling $O(n \log n)$.
  - **Packing**: The packing process involves traversal of wide rectangles taking $O(n^2)$,and choosing the narrow rectangles taking $O(1)$.
  - **Overall**: $O(n \log n) + O(n) + O(n^2) + O(1) = O(n^2)$.
- **Space Complexity**:
  - **Storage for Rectangles**: $O(n)$.
  - **Narrow and Wide Lists**: $O(n)$ each, totaling $O(n)$.
  - **Overall**: $O(n)$.

4. Advanced Size Alternating Stack (ad_SAS)

- **Time Complexity**:
  - **Partitioning**: Dividing rectangles into narrow and wide categories takes $O(n)$.
  - **Sorting**: Sorting narrow and wide rectangles takes $O(n \log n)$ each, totaling $O(n \log n)$.
  - **Packing**: The packing process involves traversal of wide rectangles taking $O(n^2)$,and choosing the narrow rectangles taking $O(1)$.
  - **Overall**: $O(n \log n) + O(n) + O(n) = O(n^2)$.
- **Space Complexity**:
  - **Storage for Rectangles**: $O(n)$.
  - **Narrow and Wide Lists**: $O(n)$ each, totaling $O(n)$.
  - **Overall**: $O(n)$.

5. Sleator Algorithm

- **Time Complexity**:
  - **Partitioning**: Dividing rectangles into groups A and B takes $O(n)$.
  - **Sorting**: Sorting group B rectangles takes $O(n \log n)$.
  - **Packing**: Placing rectangles in group A and balancing group B takes $O(n)$.
  - **Overall**: $O(n \log n) + O(n) = O(n \log n)$.
- **Space Complexity**:
  - **Storage for Rectangles**: $O(n)$.
  - **Groups A and B**: $O(n)$ each, totaling $O(n)$.
  - **Overall**: $O(n)$.
- **Approximation ratio**:
  $Sleator(I) \leq 1.5OPT(I) + hmax \leq 2.5OPT(I)$

6. Split Packing (SP)

- **Time Complexity**:
  - **Sorting**: Sorting rectangles by width and height takes $O(n \log n)$.
  - **Packing**: The algorithm iterates over rectangles and strips, leading to $O(n)$ in the worst case due to splitting strips.
  - **Overall**: $O(n \log n) + O(n) = O(n \log n)$.
- **Space Complexity**:
  - **Storage for Rectangles**: $O(n)$.
  - **Strips**: In the worst case, each rectangle could create a new strip, resulting in $O(n)$.
  - **Overall**: $O(n)$.
- **Approximation ratio**:
  $SP(I) \leq 2OPT(I) + hmax \leq 3OPT(I)$

Summary

- **Time Complexity (NOT considering sort)**:
  - FFDH: $O(n^2)$
  - NFDH: $O(n)$
  - SAS: $O(n^2)$
  - ad_SAS: $O(n^2)$
  - Sleator: $O(n)$
  - SP: $O(n)$
- **Space Complexity**:
  - All algorithms have a space complexity of $O(n)$, as they primarily use linear space to store rectangles and auxiliary data structures.

As the width factor increases, FFDH, NFDH, Sleator, SP and SAS_ad performances are largely unaffected. SAS show a greater sensitivity to changes in width.

FFDH is the most stable algorithm, and when it comes to All Narrow Cases, SP performs poorly. SAS_ad is close to NFDH when all rectangles are the same shape.

NFDH and FFDH algorithm provide excellent approximate raito and simple implement. NFDH and Sleator provide excellent efficiency.

In conclusion, the choice of algorithm depends on the specific requirements of the application, such as the distribution of texture dimensions, the width-to-height ratio, and the overall size of the texture set. Each algorithm has its strengths and weaknesses, and the selection should be based on the specific constraints and objectives of the texture packing problem.