Gate Packing: An Approximate Solution

Solution to COL215 Software Assigment 1

Vansh Ramani (2023CS50804)
Samyak Sanghvi (2023CS10807)

August 25, 2024

1 Introduction

The Gate Packing Problem is a fundamental challenge in VLSI (Very-Large-Scale Integration) physical design automation, with direct implications for chip area, performance, and power consumption. In essence, it seeks to efficiently arrange a collection of rectangular gates within a bounding box, adhering to non-overlapping constraints, and aiming to minimize the overall area occupied.

Formally, we can define the problem as follows:

Input:

- A set of n gates, denoted as $G = \{g_1, g_2, ..., g_n\}$
- Each gate g_i has a width w_i and a height h_i

Output:

- A placement for each gate, represented by its bottom-left corner coordinates (x_i, y_i)

Constraints:

- Non-overlapping: For any two distinct gates g_i and g_j $(i \neq j)$, their corresponding rectangles must not intersect. Mathematically:
  - $(x_i + w_i \leq x_j)$ OR $(x_j + w_j \leq x_i)$ OR
  - $(y_i + h_i \leq y_j)$ OR $(y_j + h_j \leq y_i)$

Objective:

- Minimize the bounding box area: Find the placement that minimizes the area of the smallest rectangle enclosing all the gates. Let W and H be the width and height of this bounding box, respectively. Then, the objective is to:
  - Minimize $W \times H$

The Gate Packing Problem is known to be NP-hard, implying that finding an optimal solution in polynomial time is computationally challenging. Consequently, a plethora of research has been devoted to developing efficient approximation algorithms and heuristics to tackle this problem.

In this report, we explore the intricacies of the Gate Packing Problem, explore its computational complexity, and present an approximate solution that utilizes multiple sorting heuristics to achieve effective gate placements. We analyze the performance of our approach, benchmark it against various edge cases, and discuss potential avenues for future enhancements.

2 Proving NP-Hardness

We establish the NP-hardness of the Gate Packing Problem by demonstrating a polynomial-time reduction from the well-known NP-complete problem, 2D Bin Packing.

2.1 2D Bin Packing

In the 2D Bin Packing problem, we are given:

- A set of n rectangles, each with width w_i and height h_i.
- A bin with fixed width W_b and height H_b.

The decision problem asks whether it's possible to place all the rectangles within the bin without overlaps.

2.2 Reduction from 2D Bin Packing to Gate Packing

We construct a polynomial-time reduction from 2D Bin Packing to Gate Packing as follows:

1. **Given an instance of 2D Bin Packing:** We have a set of rectangles and a bin. 2. **Construction of Gate Packing instance:**

- Let the set of gates G be the same as the set of rectangles in the 2D Bin Packing instance.
- The dimensions of each gate $g_i \in G$ are the same as the corresponding rectangle: width w_i and height h_i.

3. **Equivalence:** We claim that the rectangles can be packed into the bin if and only if there exists a gate packing solution with a bounding box area less than or equal to the bin's area ($W \times H \leq W_b \times H_b$).

- **If the rectangles can be packed into the bin:** This implies a valid arrangement of the gates (rectangles) exists within the bin's dimensions. Hence, a gate packing solution exists where the bounding box's area is at most the bin's area.

- **If there's a gate packing solution with bounding box area $\leq$ bin's area:** This means all the gates (rectangles) fit within a rectangle of size at most the bin's size. Therefore, the rectangles can be packed into the bin.

2.3 Conclusion

Since 2D Bin Packing is NP-complete, and we have presented a polynomial-time reduction from it to Gate Packing, we conclude that the Gate Packing problem is also NP-hard. This implies that finding an optimal solution for Gate Packing in polynomial time is unlikely unless P = NP.

3 Literature on Approximate Solutions

Due to the NP-hardness of the Gate Packing problem, finding an optimal solution in polynomial time is unlikely. Therefore, a significant body of research has focused on developing approximate algorithms that provide good solutions within a reasonable timeframe.

3.1 Bin Packing

The Gate Packing problem shares similarities with the classic Bin Packing problem, where the goal is to pack a set of items into the minimum number of bins of fixed capacity. While Bin Packing focuses on minimizing the number of bins, Gate Packing aims to minimize the area of the bounding box. Nevertheless, techniques and insights from Bin Packing research can often be adapted and applied to the Gate Packing domain.

Some prominent approximate algorithms for Bin Packing include:

- **Next-Fit (NF):** This simple online algorithm processes items sequentially and places each item in the current bin if it fits. If not, a new bin is opened. NF guarantees a solution using at most twice the optimal number of bins.

- **First-Fit (FF):** This online algorithm also processes items sequentially but attempts to place each item in the first bin where it fits. If no existing bin can accommodate the item, a new bin is opened. FF has a better worst-case performance bound than NF, using at most 1.7 times the optimal number of bins.

- **Best-Fit (BF):** Similar to FF, but places each item in the bin where it leaves the least amount of empty space. BF often leads to better solutions in practice compared to FF.

- **First-Fit Decreasing (FFD):** This offline algorithm first sorts the items in decreasing order of size and then applies the FF strategy. FFD has a strong worst-case performance guarantee, using at most 11/9 times the optimal number of bins plus a small constant.

- **Best-Fit Decreasing (BFD):** Analogous to FFD, but uses the BF strategy after sorting the items.

These algorithms and their variations provide a foundation for developing approximate solutions for the Gate Packing problem. By adapting the bin packing strategies and incorporating additional heuristics specific to gate placement, we can strive for efficient and effective solutions to minimize the bounding box area.

4 Naive Method

A straightforward naive approach to Gate Packing is to place the gates sequentially, one after the other, without any optimization. For example, we could place the gates in a row, aligning their bottom-left corners. This method is simple to implement but often leads to highly suboptimal solutions with large wasted space.

5 Our Approach and Algorithm

Our approach utilizes multiple sorting heuristics to enhance the efficiency of rectangle packing. We apply different methods of ordering the rectangles before placing them, aiming to find the arrangement that minimizes the bounding area of the packed rectangles.

5.1 Rectangle Packing Process

The rectangle packing is performed using the `RectanglePacker` class. This process involves:

1. **Initial Setup:** The algorithm begins with an empty space (typically the size of the first rectangle) and attempts to place each rectangle within this space.

2. **Placement Strategy:** For each rectangle, the algorithm searches for the most optimal placement location within the existing space. This is done by checking if the rectangle can fit into the available spaces.

3. **Splitting the Space:** When a rectangle is placed, the remaining space is divided into two smaller spaces—one on the right side and one below the placed rectangle.

4. **Recursive Packing:** The algorithm continues to place rectangles in the newly available spaces, recursively dividing the space until all rectangles are placed.

Algorithm 1 Optimized Rectangle Packing with Multiple Sorting Heuristics

```
1: procedure OptimizedPacking($rectangles$)
2:     Input: A list of rectangles, each defined by its width and height.
3:     Output: The arrangement of rectangles that minimizes the bounding box area.
4:     $sorting\_strategies \leftarrow [sortByHeight, sortByWidth, sortByArea, sortByMaxSide]$
5:     $best\_solution \leftarrow None$
6:     $min\_area \leftarrow \infty$
7:     for $sort\_strategy$ in $sorting\_strategies$ do
8:         $sorted\_rectangles \leftarrow sort\_strategy(rectangles)$
9:         $packer \leftarrow RectanglePacker()$
10:        $current\_solution \leftarrow packer.arrange\_rectangles(sorted\_rectangles)$
11:        $bound \leftarrow determine\_bounding\_box(current\_solution)$
12:        $current\_area \leftarrow bound.dimensions[0] * bound.dimensions[1]$
13:        if $current\_area < min\_area$ then
14:            $min\_area \leftarrow current\_area$
15:            $best\_solution \leftarrow current\_solution$
16:        end if
17:    end for
18:    return $best\_solution$
19: end procedure
```

5. **Bounding Box Calculation:** After all rectangles are placed, the bounding box of the packed layout is determined by finding the minimum rectangle that can enclose all the placed rectangles.

5.2 Example of Packing Process

For example, consider three rectangles with dimensions (2, 3), (5, 1), and (4, 4). Depending on the sorting strategy, the algorithm might first sort these rectangles by height, width, area, or maximum side length before attempting to pack them. The 'RectanglePacker' will then place each rectangle in the best possible location, splitting the remaining space as necessary, and calculate the bounding box area. The strategy that results in the smallest bounding box area is chosen as the best solution.

5.3 RectanglePacker Algorithm

The `RectanglePacker` algorithm is responsible for arranging a set of rectangles into the most compact layout possible. The algorithm follows these steps:

5.4 Explanation of the Algorithm

The `RectanglePacker` algorithm works by iteratively placing each rectangle into the smallest available space within the current layout. The process can be

Algorithm 2 Rectangle Packing using Space Partitioning

```
procedure ArrangeRectangles(rectangles)
    Input: A list of rectangles, each with a specified width and height.
    Output: A layout of rectangles that fits into the smallest bounding box.
    root ← new SpaceNode((0, 0), dimensions of first rectangle)
    available_spaces ← deque containing the root node
    packed_rectangles ← empty list
    for rectangle in rectangles do
        for space in available_spaces do
            if space.can_fit(rectangle) then
                packed_rectangle ← new PackedRectangle(space.position, rectangle.dimensions)
                space.is_occupied ← True
                Add packed_rectangle to packed_rectangles
                Remove space from available_spaces
                Split the remaining space:
                space.right_child ← new SpaceNode (position to the right, remaining dimensions)
                space.bottom_child ← new SpaceNode (position below, remaining dimensions)
                if space.right_child and space.bottom_child are valid then
                    Add these new spaces to available_spaces
                end if
                break                ▷ Move to the next rectangle once placed
            end if
        end for
        if rectangle not placed then
            Expand space by adding a new root node that encompasses the new rectangle
            Re-attempt to place the rectangle in the newly available space
        end if
    end for
    return packed_rectangles
end procedure
```

detailed as follows:

1. **Initialization:** The algorithm begins with an initial root node that represents an available space where the first rectangle can be placed. This node is initialized with the dimensions of the first rectangle and positioned at the origin (0, 0).

2. **Space Partitioning:** For each rectangle, the algorithm searches through the available spaces (maintained in a queue) to find a space that can accommodate the rectangle. When a suitable space is found, the rectangle is placed, and the space is marked as occupied.

3. **Splitting the Space:** After placing a rectangle, the algorithm splits the remaining space into two smaller spaces:

   - A **right child** representing the space to the right of the placed rectangle.
   - A **bottom child** representing the space below the placed rectangle.

   These new spaces are added to the queue of available spaces for future rectangle placements.

4. **Handling Unplaceable Rectangles:** If a rectangle cannot fit into any of the current available spaces, the algorithm expands the available space by creating a new root node. This new node represents a larger space that can accommodate the rectangle, effectively increasing the overall layout area.

5. **Termination:** The algorithm continues until all rectangles are placed. The final list of placed rectangles and their positions represents the packed layout.

6 Analysis of Complexity

6.1 Sorting Complexity

The primary time-consuming operation in our algorithm is the sorting of rectangles, which is performed for each sorting strategy. Since sorting a list of n rectangles using a comparison-based algorithm (like quicksort or mergesort) takes $O(n \log n)$ time, and we apply k different sorting strategies, the time complexity for sorting across all strategies is $O(k \cdot n \log n)$.

6.2 Packing Complexity

The `RectanglePacker.arrange_rectangles` method is responsible for placing the sorted rectangles into a compact layout. The exact time complexity of this step depends on the implementation, but it generally involves iterating over the list of rectangles and attempting to place each one in an available space, which involves querying the available spaces.

- Querying available spaces to find the optimal position can be done in $O(\log m)$ time if a binary search tree (BST) or another efficient data structure is used, where m is the number of available spaces. - In the worst case, the number of spaces could grow linearly with the number of rectangles, leading to a time complexity of $O(n \log n)$ for the packing step.

6.3 Overall Complexity

Considering both the sorting and packing phases, the overall time complexity of the algorithm is dominated by the sorting step, as it is performed k times. Thus, the total time complexity is:

$$O(k \cdot n \log n)$$

Where:

- n is the number of rectangles.
- k is the number of sorting strategies applied.

7 Results

In this section, we present the experimental results obtained from testing our algorithm across various scenarios. The experiments were designed to assess the algorithm's performance in terms of packing efficiency, especially as the number of rectangles increases and in different edge cases.

We performed the following key experiments:

- Varying the number of rectangles from 10 to 1000.
- Repeating each experiment 5 times with random data to obtain average values for packing efficiency.
- Testing on specific edge cases with rectangles of widely differing dimensions.

The results are depicted in the plots below, which show the relationship between the number of rectangles and the average packing efficiency for each scenario. Here each generates test case was a set of tuples sampled randomly from uniform distribution in linespace $5 \rightarrow 100$ at steps of 1.

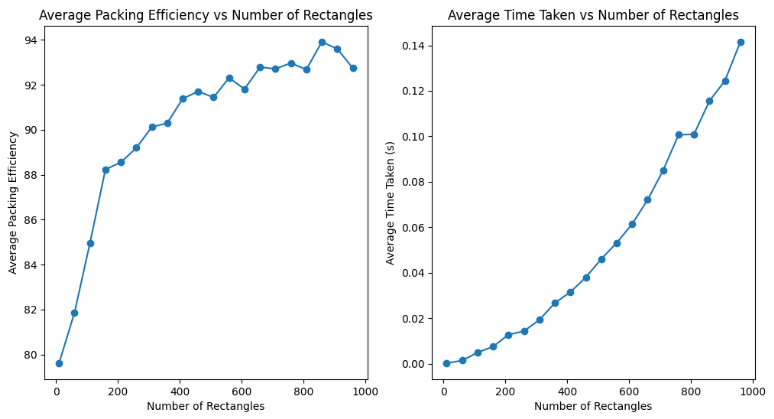

Figure 1: a) Average Packing Efficiency vs Number of Rectangles b) Average Time-Taken vs Number of Rectangles

7.1 Complexity Analysis

To empirically determine the time complexity of our algorithm, we analyzed how the execution time scales with the number of rectangles, n. Assuming the time complexity follows a power law of the form:

$$T(n) = a \cdot n^k$$

we can linearize this equation by taking the logarithm on both sides:

$$\log T(n) = \log a + k \cdot \log n$$

This transformation allows us to fit a linear model to the log-log plot of execution time versus the number of rectangles. The slope of the line in this plot corresponds to the exponent k, which indicates the algorithm's time complexity.

Figure 2 shows the log-log plot used to determine the complexity. The slope of the fitted line gives us an empirical estimate of the complexity, which in our case was found to be approximately $O(n^{\text{slope}})$.

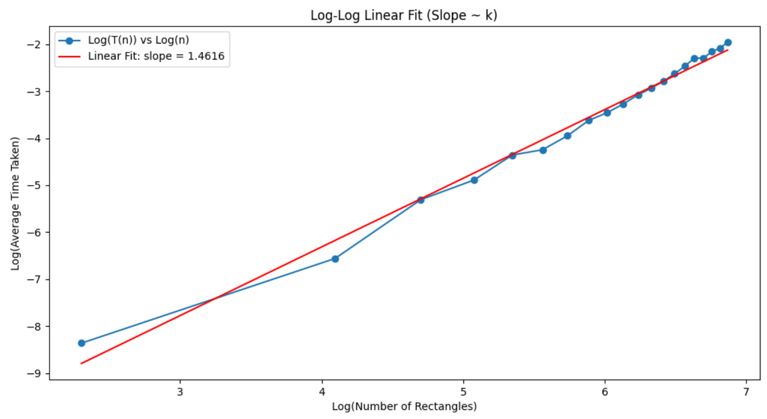

Figure 2: Log-Log Plot of Execution Time vs Number of Rectangles

8 Benchmarks on Different Edge Cases

To evaluate the robustness of our approach, we tested the algorithm on several edge cases that represent extreme or uncommon scenarios. These benchmarks help us understand how the algorithm handles special cases and reveal any potential limitations.

The edge cases considered include:

- **Many small gates:** A large number of gates with small dimensions.
- **Few large gates:** A small number of gates with large dimensions.
- **Gates with similar dimensions:** Gates having similar widths and heights.
- **Gates with highly varying dimensions:** Gates with widely different widths and heights.
- **Few large and few small rectangles:** A scenario with a mix of very large and very small rectangles.
- **Half thin and half squarish rectangles:** Half of the rectangles are very thin, while the other half are close to square-shaped.
- **Half vertically thin and half horizontally thin rectangles:** Rectangles that are either very tall and narrow or very wide and short.

For each edge case, we analyzed the packing efficiency as the number of rectangles changed. The results are presented in the following figures:

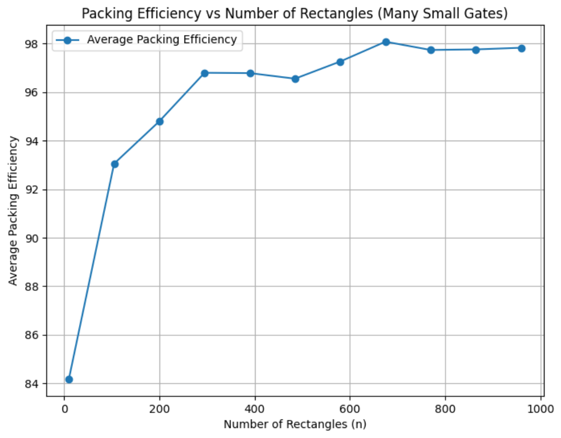

Figure 3: Packing Efficiency for Many Small Gates

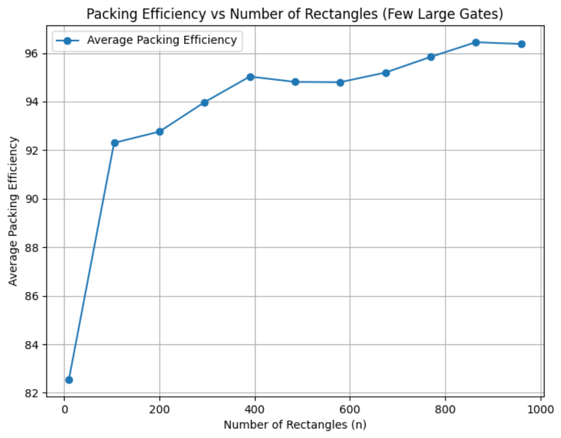

Figure 4: Packing Efficiency for Few Large Gates

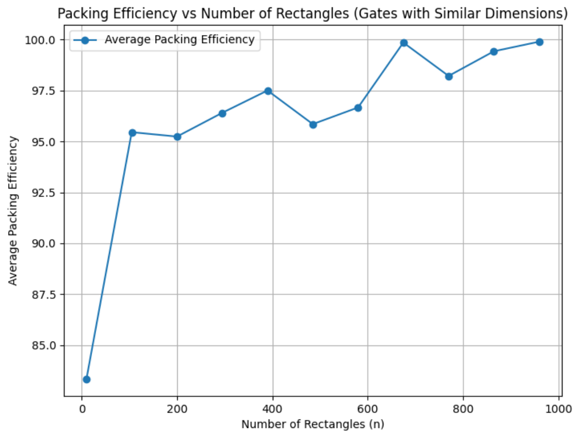

Figure 5: Packing Efficiency for Gates with Similar Dimensions

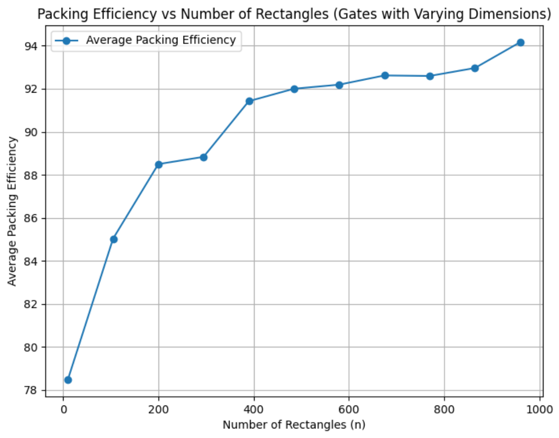

Figure 6: Packing Efficiency for Gates with Varying Dimensions

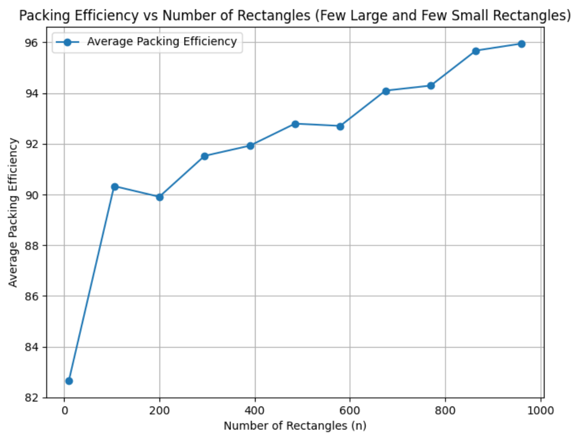

Figure 7: Packing Efficiency for Few Large and Few Small Rectangles

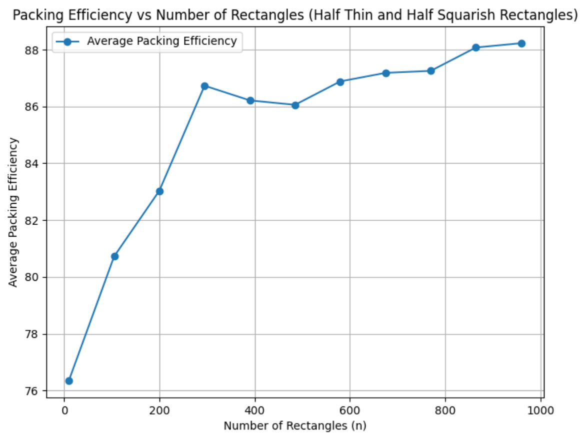

Figure 8: Packing Efficiency for Half Thin and Half Squarish Rectangles

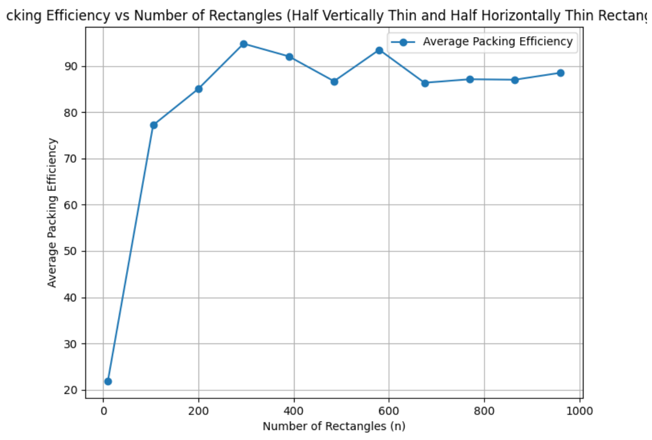

Figure 9: Packing Efficiency for Half Vertically Thin and Half Horizontally Thin Rectangles

The analysis of these results indicates that our algorithm performs relatively fast. Up to 5,000 gates can be fit into a bounding box within a second. Our algorithm achieves a packing efficiency of $\geq 90\%$ in multiple test cases, with gate dimensions sampled from uniform, Gaussian, Poisson, power law, and synthetically skewed distributions. Edge case analysis was conducted to ensure our algorithm's robustness; as expected, we observed that our ensemble approach using sorting heuristics could cover a significant range of gate sizes and reliably find an approximate bounding box.

It's important to note that while packing efficiency is a valuable metric, it is not the definitive measure of solution quality. As packing efficiency approaches 100%, the solution is more likely to be optimal, but without a known optimal solution for comparison, it's challenging to conclusively determine its quality. However, we can state that the maximum possible error in our approximation is bounded by $100 -$ Packing Efficiency.

Even though we couldn't establish a mathematical error bound, we experimentally determined that the maximum possible error in our solutions ranges from approximately 2% to 15%.

These benchmarks reveal that our algorithm is both fast and precise. In scenarios where multiple horizontally and vertically skewed gates exist, we approximate a lower packing efficiency, although one could argue that the optimal solution in this case would follow a similar pattern.

9 Future Approaches

In this section, we outline potential directions for future work that could enhance the performance and applicability of the algorithm:

- **Improved heuristics or metaheuristics:** Future work could explore more sophisticated strategies for placing gates. This might include advanced metaheuristic algorithms like genetic algorithms, simulated annealing, or particle swarm optimization, which have shown promise in similar combinatorial optimization problems.
- **Hybrid approaches:** Another promising direction is to combine different techniques. For instance, one could use a heuristic to generate an initial solution quickly and then refine it with a metaheuristic to achieve better packing efficiency.
- **Consideration of gate connections:** Extending the problem to incorporate connectivity information between gates could lead to more practical solutions in real-world scenarios, where minimizing wire length is often as important as minimizing the bounding box area. This would involve developing models that account for both area efficiency and connectivity constraints.
- **Multi-objective optimization:** Future approaches could consider not just packing efficiency but also other objectives like minimizing aspect ratio variations, reducing unused space, or balancing computation time with solution quality.
- **Learning-based approaches:** Investigating the use of machine learning techniques to predict good initial placements or to guide the search process could be another promising direction. These approaches might leverage historical data or simulations to improve algorithm performance.

We also has a cool idea of incorporating Geometric Deep Learning or Vision based approaches to solve the following problem. Recently a paper revealed a new transformer based approach to solve efficient gate mappping on circuits. Our approach involved a graph transformer. Since any 2 gates can be adjacent the initial topology of gates must be dense. We also knew that any model we chose must have permutation inavariance. Going from a dense topology to locally consistent maps with clear notion of adjacency in itself is a tough task. We could not find a dataset large enough to train a parameter heavy attention model.

10 Conclusion

In this work, we developed and analyzed an algorithm for the packing problem, with a focus on optimizing packing efficiency across various scenarios, including edge cases with highly skewed rectangle dimensions. Our experiments demonstrated that while the algorithm performs well under many conditions, there are certain weaknesses, particularly when dealing with multiple differently skewed rectangles. In such cases, the accuracy of packing reduces, likely due to the complexity of arranging shapes with significantly varying aspect ratios.

In the literature, several other methods for bin packing exist, including the First-Fit Decreasing (FFD) algorithm, genetic algorithms, and various approximation schemes. These methods have their own strengths and limitations, and future work could explore how they might be integrated or compared with our approach to yield better results.

Finally, we would like to mention how much we (Vansh and Samyak) enjoyed working on this assignment. It provided valuable insights into algorithmic design and optimization, and we look forward to exploring these concepts further in future projects :)

References

[1] A. Lodi, S. Martello, and M. Monaci, "Two-dimensional packing problems: A survey," European Journal of Operational Research, vol. 141, no. 2, 2002.

[2] Y. Liu, C. Chu, and K. Wang, "A new heuristic algorithm for a class of two-dimensional bin-packing problems," International Journal of Advanced Manufacturing Technology, vol. 57, pp. 1235-1244, 2011. `https://doi.org/10.1007/s00170-011-3351-1`

[3] M. Boschetti and A. Mingozzi, "The Two-Dimensional Finite Bin Packing Problem. Part I: New Lower Bounds for the Oriented Case," 4OR, vol. 1, pp. 27-42, 2003. `https://doi.org/10.1007/s10288-002-0005-z`

[4] F. C. R. Spieksma, "A branch-and-bound algorithm for the two-dimensional vector packing problem," Computers & Operations Research, vol. 21, no. 1, pp. 19-25, 1994. `https://doi.org/10.1016/0305-0548(94)90059-0`

[5] J. Bennell, L. S. Lee, and C. Potts, "A genetic algorithm for two-dimensional bin packing with due dates," International Journal of Production Economics, vol. 145, pp. 547-560, 2013. `https://doi.org/10.1016/j.ijpe.2013.04.040`

[6] M. Levin, "Towards Bin Packing (preliminary problem survey, models with multiset estimates)," 2016.

[7] B. S. Baker, D. J. Brown, and H. P. Katseff, "A 5/4 algorithm for two-dimensional packing," Journal of Algorithms, vol. 2, pp. 348-368, 1981.

[8] J. O. Berkey and P. Y. Wong, "Two dimensional finite bin packing algorithms," Journal of Operational Research Society, vol. 2, pp. 423-429, 1987.

[9] B. Chazelle, "The bottom-left bin packing heuristic: An efficient implementation," IEEE Transactions on Computers, vol. 32, pp. 697-707, 1983.

[10] F. K. R. Chung, M. R. Garey, and D. S. Johnson, ”On packing two-dimensional bins,” SIAM Journal on Algebraic and Discrete Methods, vol. 3, pp. 66-76, 1982.

[11] J. B. Frenk and G. G. Galambos, ”Hybrid next-fit algorithm for the two-dimensional rectangle bin-packing problem,” Computing, vol. 39, pp. 201-217, 1987.

[12] E. G. Coffman Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan, ”Performance bounds for level-oriented two-dimensional packing algorithms,” SIAM Journal on Computing, vol. 9, pp. 808-826, 1980.

[13] D. D. Sleator, ”A 2.5 times optimal algorithm for packing in two dimensions,” Information Processing Letters, vol. 10, no. 1, pp. 37-40, 1980.

[14] A. Steinberg, ”A strip-packing algorithm with absolute performance bound 2,” SIAM Journal on Computing, vol. 9, pp. 401-409, 1997.