Report

Name: 張硯博 **Student ID**: 113062645

1. Introduction

The project requires implementing a program that inserts as many staples as possible into the layout, ensuring no staggering and a balanced number of VDD-to-VDD and VSS-to-VSS staples. To adapt the method proposed in [1], several challenges must be addressed. The most critical issue is the time constraint: this project has a 600-second time limit, whereas the most time-consuming test case in [1] takes approximately 3600 seconds, as shown in their experimental results. To tackle this problem, I decided to implement a two-stage heuristic algorithm and use multithreading to accelerate the computation and do not consider flipping the cell.

---

2. Approach

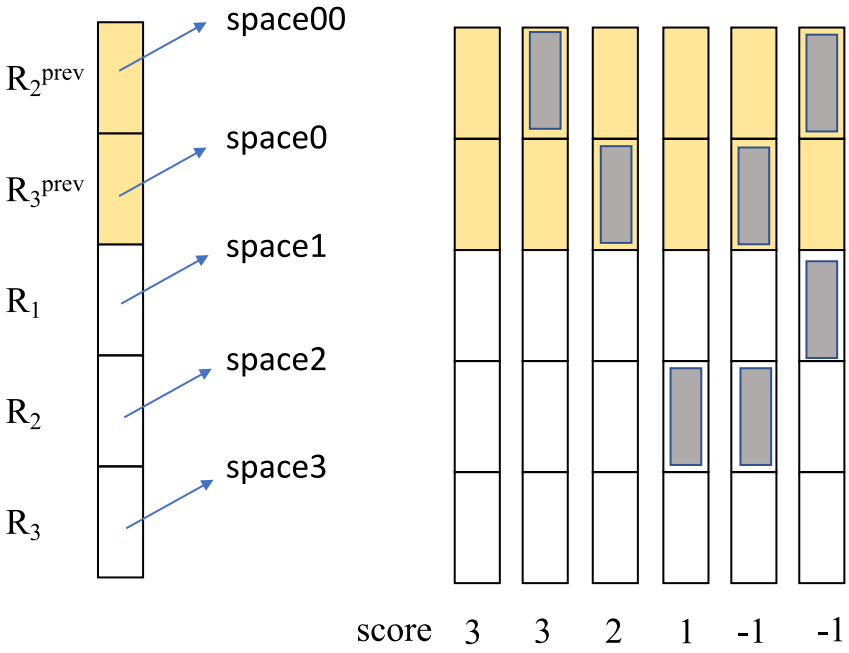
Instead of finding new positions for the cells and inserting staples simultaneously, I split the process into two stages for the sake of speed, reduced memory usage, and easier implementation as mentioned in the introduction.

2.1 Stage1

The first stage aims to find new positions for the cells such that the number of continuous blank spaces is maximized. I adopt the dynamic programming (DP) framework proposed in [1], which considers three rows at the same time. Each DP state is defined by seven arguments: cur, s1, l1, s2, l2, s3, and l3. The definition of these arguments and the transition method are the same as in [1]; The only difference lies in the way of updating benefit. I use a new evaluation function designed to maximize continuous blank spaces heuristically.
Let the variable $spacei, 1 \leq i \leq 3$ be true if Ri is empty, false otherwise, $space0$, and $space00$ be true if R_3^{prev}, and R_2^{prev}, respectively, is empty, otherwise false.The

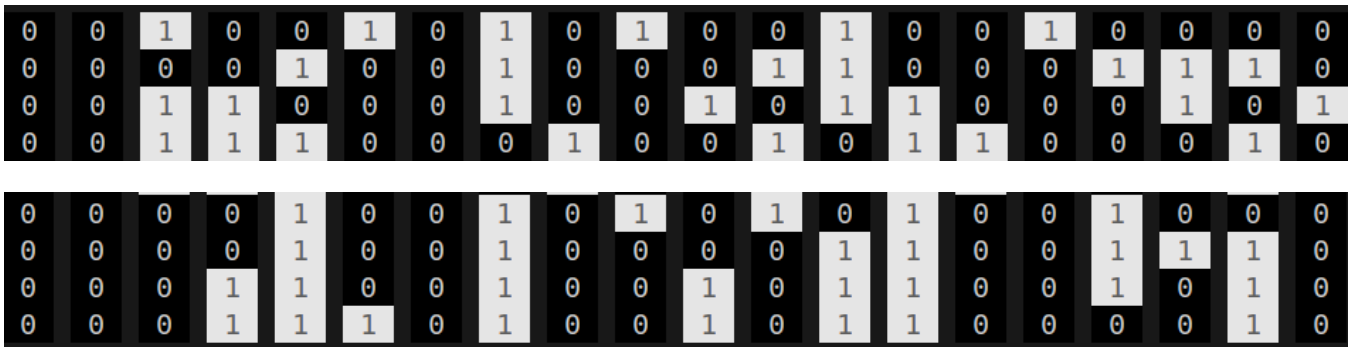
evaluation rule is as follows: (the figure below shows some examples of evaluation)

```
if(space0 && space1) added+=1;
if(space1 && space2) added+=1;
if(space2 && space3) added+=1;
if(!space00 && space0 && !space1) added -= 1;
if(!space0 && space1 && !space2) added -=1;
if(!space1 && space2 && !space3) added -=1;
```



To implement the DP, instead of using a bottom-up approach and constructing a graph where each node represents a state, I adopt a top-down recursive approach with memoization for easier implementation. Regarding memory constraints—since it's impractical to declare an array with so many dimensions—I use a hash table (`unordered_map`) to store the results of each state and the best result obtained from recursive calls. Since the standard `unordered_map` does not provide a hash function for arrays, I referenced the hash function proposed in [2]. After completing the recursion, I use the recorded map to backtrack and retrieve the updated positions of all the cells.

The two figures below show the layout before and after applying the DP algorithm, where 0 denotes a blank space and 1 denotes a pin. As can be seen, the evaluation effectively increases the amount of continuous blank space and concentrates the pins, making the layout easier for staple insertion.

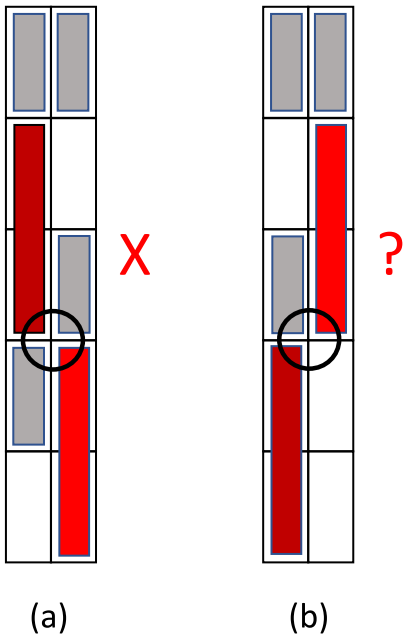


2.3 Parallelizing Stage 1 with 8 Threads

For testcases with large layouts and a high number of cells, the runtime may exceed the time limit. To address this, I divide the layout into 8 non-overlapping blocks, each containing a group of continuous rows, and use 8 threads to parallelize the computation. The choice of 8 threads is based on memory considerations: a single 3-row DP instance consumes approximately 1 to 2 GB of memory for large testcases. Although using more threads could further reduce runtime, it would cause the program’s memory usage to exceed the 16GB limit. With this approach, all testcases except the largest one can be completed within the constraints. For the largest testcase, I apply a cost down method, where, with a 50% chance, I reduce the maximum shifting distance of a cell by one site to lower the complexity.

2.2 Stage2

After Stage 1, I greedily insert staples into the layout column by column from left to right, and within each column from top to bottom. This step is much easier than the previous one; the main challenge is ensuring proper staggering checks. The figure below illustrates how to handle the two types of staggering.



If case (a) is encountered, the bright red staple cannot be inserted.
However, in case (b), there may be a chance that the bright red staple can be inserted. This is because if a staple can be inserted directly below it, then staggering does not occur. Therefore, I recursively check whether the bottom part can be inserted.

The figure below shows the result after insertion, where 2 denotes the top half of a staple and 3 denotes the bottom half.

As for balancing the number of VDD-to-VDD and VSS-to-VSS staples, the three-row DP naturally produces a balanced result, as mentioned in [1]. Except for the first test case, which has a very small layout and results in imbalanced staples, all other test cases naturally generate balanced staples. In cases where imbalance does occur, I implement a check function that greedily removes staples until the balance is restored.

---

3. Time and Space Complexity

It is clear that the bottleneck of this program lies in Stage 1, so I analyze the

complexity of Stage 1 only.

Both the time complexity and space complexity depend on the number of states generated in the DP. Since the DP framework is the same as in [1], the complexity is also the same as reported in [1], which is $O(x_{\max} \cdot s_{\max}^3 \cdot (2\Delta x + w_{\max} + e_{\max})^3)$. The definitions of these symbols can be found in section 3.1 of [1].

---

4. Experiment result

The figure below shows the experimental results for different testcases. For testcases 0, 1, and 2, the layout sizes are small, so using a single thread is sufficient and even results in more staples. Therefore, I use only one thread for these cases. For the other test cases, 8 threads are used. As shown, both runtime and memory usage increase with the size of the testcase. Note that the last test case has been trimmed, as mentioned in the approach section, to meet the time and memory constraints. As a result, its memory usage is lower than that of testcase 6.

Testcase	Staple Size	Ratio	Runtime (s)	Memory	Number of Threads
0	1,648	1.09936	9.042	< 1GB	1
1	13,227	1.05612	63.691	< 1GB	1
2	6,914	1.08819	77.642	< 1GB	1
3	90,403	1.02720	91.333	5GB	8
4	59,970	1.00696	104.174	5.7GB	8
5	201,555	1.07223	170.688	6.9GB	8
6	227,075	1.01896	431.172	10.9GB	8
7	832,603	1.02098	532.219	10.3GB	8

---

5. References

[1] Y. -J. Xie, K. -Y. Chen and W. -K. Mak, "Manufacturing-Aware Power Staple Insertion Optimization by Enhanced Multi-Row Detailed Placement Refinement," 2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)

[2] https://stackoverflow.com/questions/20511347/a-good-hash-function-for-a-vector