CENG 4120 Computer-Aided Design of Very Large Scale Integrated Circuits

Homework 2 (Part B)
Due data: March 22, 2025 11:59PM

1 Introduction

We have introduced slicing floorplanning in Lecture 9. The key idea is of slicing floorplanning is to recursively divide a rectangle into two with a vertical or horizontal cutline. Then, we can represent the slicing floorplaning result with a binary tree with leaves denoting rectangles and internal nodes representing the cutlines. Furthermore, we can a Polish expression to represent this tree (and the original floorplanning). In this homework, We will go in the opposite direction, given each rectangle's alternative shapes and the Polish expression, find the feasible slicing floorplanning with the smallest area and compute coordinates of each rectangle.

At the end of this document, some additional knowledge will be provided for reference when you encounter problems in this homework.

2 Problem

2.1 Input

Your program should read from a text file like:

```
3
10 10
7 4
8 2
0 1 2 + *
```

The first line is the number of rectangular modules n in this slicing floorplanning result. Next come n lines, each containing the width and height of the rectangular module. In the final result, each rectangular module can be rotated 90°. For example, a rectangular module with a width of 8 and a height of 2 is available in two shapes: 8×2 and 2×8. The last row is the Polish expression of this slicing floorplanning:

1. Each number in the Polish expression i represent the $(i+1)$-th rectangular module above. Numbers and operators '+' '*' are sepereated by a space.
2. '+' represents a horizontal cutline. A Polish expression like (Expression A) (Expression B) + means module A is below the cutline, while module B is above the cutline.

3. ‘*’ represents a vertical cutline. A Polish expression like (Expression A) (Expression B) * means module A is on the left side of cutline, while module B is on the right side of the cutline.

4. The aforementioned “above” and “below” refer to the directions in which the y-axis values are larger and smaller, and the “left” and “right” refer to the directions in which the x-axis values are larger and smaller.

2.2 Output

Your program should write to a text file like:

```
(0 0) (10 0) (0 10) (10 10)
(10 0) (17 0) (10 4) (17 4)
(10 4) (18 4) (10 6) (18 6)
180
```

In the first n lines, the *i*-th line represents the coordinates of each vertex of the *i*-th module in the input. Each coordinate is expressed as (x, y), and the coordinates are separated by a space. The last row is the area of this slicing floorplanning. To ensure the uniqueness of the output:

1. This slicing floorplanning should have the smallest bounding rectangle area among all feasible solutions.

2. The coordinates of each module are output in the order of lower left corner, lower right corner, upper left corner, and upper right corner.

3. Each module should be placed as far to the left and bottom as possible.

3 Grading scheme

In this homework, we will provide you with three test cases for you to verify the correctness of your program. You are encouraged to generate more test cases by yourself. There will be 7 hidden cases as well. For all test cases, $1 \leq n \leq 100$, the width w and height h of each module fulfill $1 \leq w, h \leq 100$ (there will be some big hidden cases). Your grade will depend on how many of the 10 test cases your program can pass within 20 seconds.

4 Additional Requirements

1. All of your submissions should be packaged in a single zip file.

2. Your program should be implemented using C/C++ or Python and make sure it can be compiled (for C/C++) and executed on a Linux system. If you want to use other languages, consult the TA first.

3. Apart from the source code, you should also include a README file in your submission specifying how to compile (for C/C++) and run your program.

4. Suppose the name of your executable is "hw2" (for C/C++) or "hw2.py" (for Python), it should be correctly executed using the following command in the command-line environment:

   ```
   (for C/C++) ./hw2 <in_file> <out_file>
   (for Python) python hw2.py <in_file> <out_file>
   ```

   `<in_file>` and `<out_file>` specify the paths of the input file and output file. For those who are not familiar with command-line argument processing, you can use the following code snippets.

   For C/C++:

   ```
   int main(int argc, char * argv[])
   {
       if (argc < 2 || argc > 3) {
           printf("usage: hw2 <in_file> <out_file>");
           exit(1);
       }
       char * inFile = argv[1];
       char * outFile = argv[2];
   }
   ```

   For Python:

   ```
   import sys

   if len(sys.argv) < 2 or len(sys.argv) > 3:
       print("usage: python hw2.py <in_file> <out_file>")
       exit(1)
   in_file, out_file = sys.argv[1], sys.argv[2]
   ```

5 Appendix

As a workflow for reference, this operation can be decomposed as the following steps:

1. Recover the slicing tree from the Polish expression.

2. The shape curve of each module is calculated in a bottom-up way, then get the smallest area among all feasible solutions.

3. According to the shape curve, the actual shape of each module is determined in a top-down way and the coordinates are calculated.

5.1 Handle Polish Expression

We can use the stack data structure to process Polish expression. Let's discuss an example from Lecture 9 as shown in Fig 1.

We push each element of the Polish expression onto the stack from left to right:

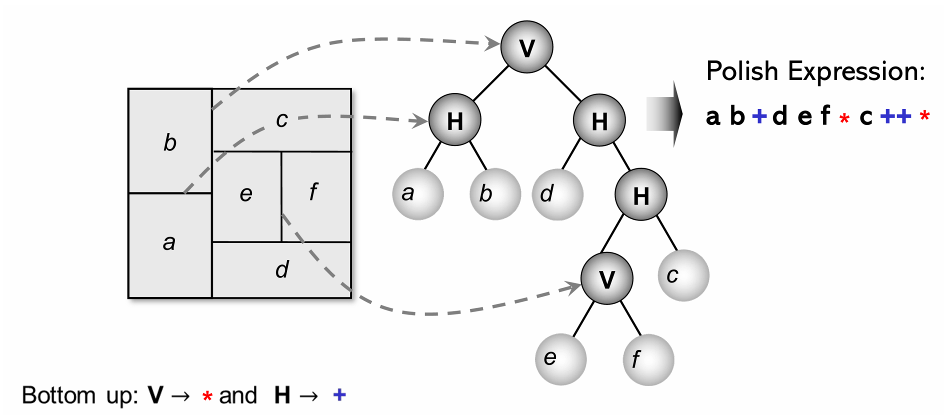

Figure 1: A PE example in Lecture 9

1. According to the properties of Polish expression, when the top element of the stack is + or *, the two subtrees of this node should already be in the stack, or more precisely, they should be directly below the top element of the stack. For example:

   - When the stack contains “ab+”, we can combine a and b into a larger parent module [ab+]
   - When the stack contains “[ab+]def*”, we can combine e and f into a larger parent module [ef*] while leaving d in the stack, which is because at this moment, the subtree that forms a larger module with d has not been processed.

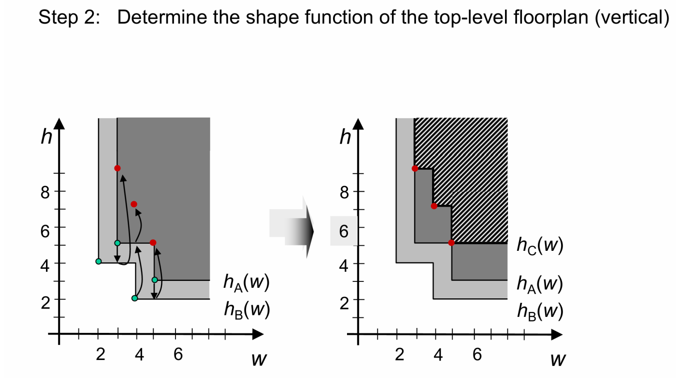

Figure 2: A shape curve example in Lecture 9

5.2 Shape Curve

In order to calculate the minimum area of the entire design, we need to find the shape curve of the top-level module. By checking the vertices of this shape curve, we can know its minimum area. In order to calculate this shape curve, we need to know the shape curves of its two children modules, so this is a bottom-up process. We can design our recursive algorithm like this:

```
void compute_shape_curve(Module& m) {
    compute_shape_curve(m.left_child);
    compute_shape_curve(m.right_child);
    // compute the shape curve of current module m ...
}
```

Let's look at the example in Lecture 9 (demonstrated in Fig 2) to see this process more closely. Here, the two children modules are divided by a horizontal cutline and will be vertically combined. The process of computing new shape curve is as follows:

- We check all green points (vertices of the shape curves of two children modules) along x-axis (w.r.t. w-axis in the figure).
- For the x value of this green point, we check if there is a corresponding definition on the shape curve of another child module. If there is, we add the two y values together to get a new y value. The x value and the new y value are combined as a vertex on the new shape curve.