B. TECH. PROJECT

On

Pattern Identification and Compression of data for Fast and Uniform Polygon Fracture

BY

Nitesh Kumar Singh

DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY INDORE

November 2017

Pattern Identification and Compression of data for Fast and Uniform Polygon Fracture

A PROJECT REPORT

Submitted in partial fulfilment of the
requirements for the award of the degrees of
BACHELOR OF TECHNOLOGY
In
COMPUTER SCIENCE AND ENGINEERING

Submitted by:
Nitesh Kumar Singh, 140001020,
Discipline of Computer Science and Engineering

Guided by:
Dr. Abhishek Srivastava
Associate Professor
Computer Science and Engineering,
IIT Indore

INDIAN INSTITUTE OF TECHNOLOGY INDORE

November 2017

CANDIDATE'S DECLARATION

I hereby declare that the project entitled **"Pattern Identification and Compression of data for Fast and Uniform Polygon Fracture"** submitted in partial fulfilment for the award of the degree of Bachelor of Technology in 'Computer Science and Engineering' completed under the supervision of **Dr. Abhishek Srivastava, Associate Professor, Computer Science and Engineering, IIT Indore** is an authentic work. Further, I declare that I have not submitted this work for the award of any other degree elsewhere.

Nitesh Kumar Singh

(140001020)

CERTIFICATE by BTP Guide

It is certified that the above statement made by the student is correct to the best of our knowledge.

Dr. Abhishek Srivastava
Associate Professor,
Discipline of Computer Science and Engineering
IIT Indore

Preface

This report on **“Pattern Identification and Compression of data for Fast and Uniform Polygon Fracture”** is prepared under the guidance of Dr. Abhishek Srivastava, Associate Professor, Computer Science and Engineering, IIT Indore.

Through this report I have tried to give a detailed description of our approach to identify the repeating pattern in polygon layout by compressing the data and I have also tried to explore the possibility of which mapping method is better to use for mapping the rectangles in unique characters. The algorithm used is verified for unit test as well as for large input cases.

I have tried to the best of my abilities and knowledge to explain the content in a lucid manner. I have also added Tables, figures,screenshots and histograms to make it more illustrative and enable the readers to understand the solution easily.

Nitesh Kumar Singh
B.Tech. IV Year
Discipline of Computer Science and
Engineering IIT Indore

Acknowledgements

I wish to thank Dr. Abhishek Srivastava for their kind support throughout the duration of the project, giving me an opportunity to work at my own pace along my own lines, while providing me with very useful directions whenever necessary.

I would also like to thank all the faculty members of Discipline of Computer Science and Engineering for their invaluable support and constructive feedback during the presentations. It is their help and support, due to which I was able to complete the design and technical report.

Finally, I offer my sincere thanks to everyone else who knowingly or unknowingly helped me complete this project.

Nitesh Kumar Singh
B.Tech. IV Year
Discipline of Computer Science and Engineering
IIT Indore

Abstract

In Electronic Design Automation industry, mask pattern is first fractured into basic rectangles, and then fabricated by the variable shaped beam mask writing machine. Ideally, mask fracture tools (EDA tools) aim at suppressing the rectangle count in order to speed up the writing time, minimizing the fracture time by compressing the data .If we will able to identify the repeating pattern of rectangles/polygon we can store the data of its fracture so that we don't have to fracture it again and again hence it reduces the rectangle count for fracturing the data.

This Project was divided in three part. First Part is Converting Non Monotone Polygons to Monotone Polygons then mapping of the rectangle in unique characters and generation of substring of unique character divided by separator. In last part the substring of unique characters are passed through a compression algorithm where data is compressed, which is stored in a result string, this string can be decompressed to get the original string and a dictionary is formed which give the result of data repetition of substring.

The main aim of the project was to check whether it is possible to convert a Non-Monotone Polygon to monotone polygon or not and know how much value for bit mapping is good enough to map the rectangle in unique character and in result of project we can see that by doubling the mapping size we can map approx. (10^5 times) more rectangles and this is what we needed. Although the time taken by increasing the mapping size is bit more but in comparison to the rectangle covered by increasing the mapping size too much in comparison to time increase.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Electronic Design Automation (EDA) is concerned with the design and production of VLSI systems. EDA systems support description of hardware at various levels of abstraction. Fig 1.1 shows various steps of the VLSI design process. Today, EDA tools have enabled the designers to complete the Design to Silicon cycle of chip manufacturing very efficiently. It enables them to work progressively down from an abstract level of design to the layout level. A layout is a complete geometric representation (i.e. **a set of Polygons**) of the masks which define how the individual layers of the circuit are to be produced. In optical lithography, light emitted from the illumination system is transmitted through the mask, and replicates the mask pattern on the wafer. During Mask Data Preparation (MDP) process, the Polygon pattern is initially **fractured** into numerous rectangles. Subsequently, these rectangles are exposed by the Variable Shaped Beam (VSB) mask writing machine. We work at the Physical Design Verification step of the VLSI design cycle, where layouts are checked for Design Rule violations set by the user.

When we Fracture a whole Polygon layout into rectangles we observe that so many rectangles are repeating similarly in different polygons which we Fracture again and again which takes time so by using Pattern detection technique we can fracture a similar kind of repeating rectangle and store its data and instead of fracturing the same repeating again we can just use the previous data to speed up the fracturing technique and instead of using whole data gain we can use compressed data because if there is redundancy of data we can compress it.

The reasons for using this Pattern Identification is two-fold. First, pattern matching's ability to easily describe very complex relationships between geometries across multiple layers simultaneously allows it to efficiently compress the design.

Second, pattern matching's true power is on display when it is integrated with other physical verification analysis or design tools. Why? Combining pattern matching with traditional design and

verification tools enables design and verification engineers to use an automated process to find areas of interest, then make design modifications and strengthen the layout against manufacturing defects, etc. This integration and automation has enabled the industry to create a breadth of new pattern detection applications that are achieving results that simply weren't possible previously.

Verification based on pattern Identification not only makes it easier to express the design rules, but the whole physical verification definition process also gets simpler and moves faster.

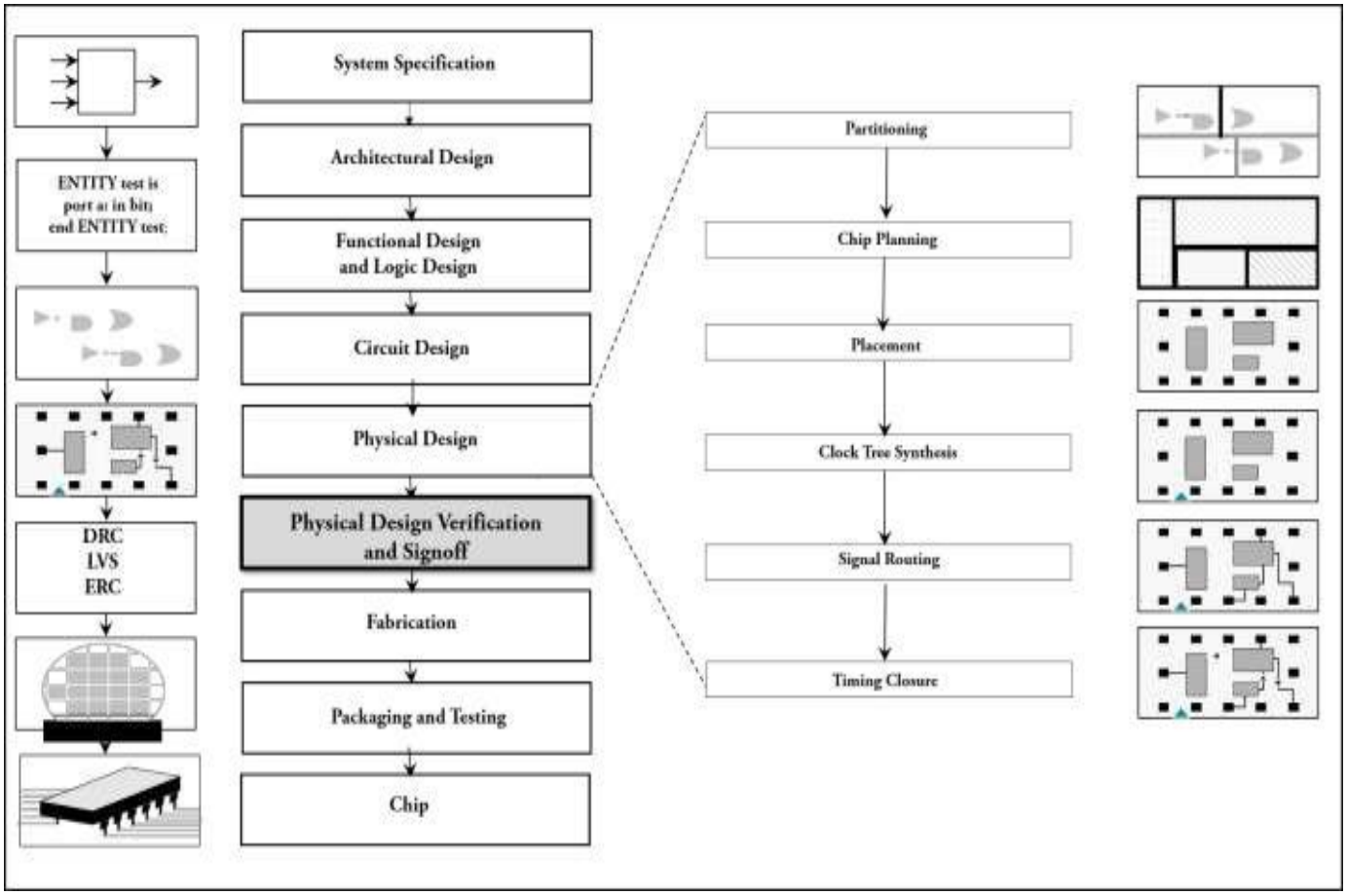


Fig 1.1 VLSI Design Flow

1.2 Objective

Given an input of millions of polygons which have to be fractured in different types of rectangle and in these pattern there is pattern of polygon which repeat to form different types of design. In order to compress the input data we have to detect these repeating pattern and store their fracture data so that they can be used if we see the same pattern again. So that we don't have to fracture same pattern again and again which will reduce execution time and faster fracture of the Polygon Layout.

Fig 1.2 will show how the Pattern will repeat in polygon layout.

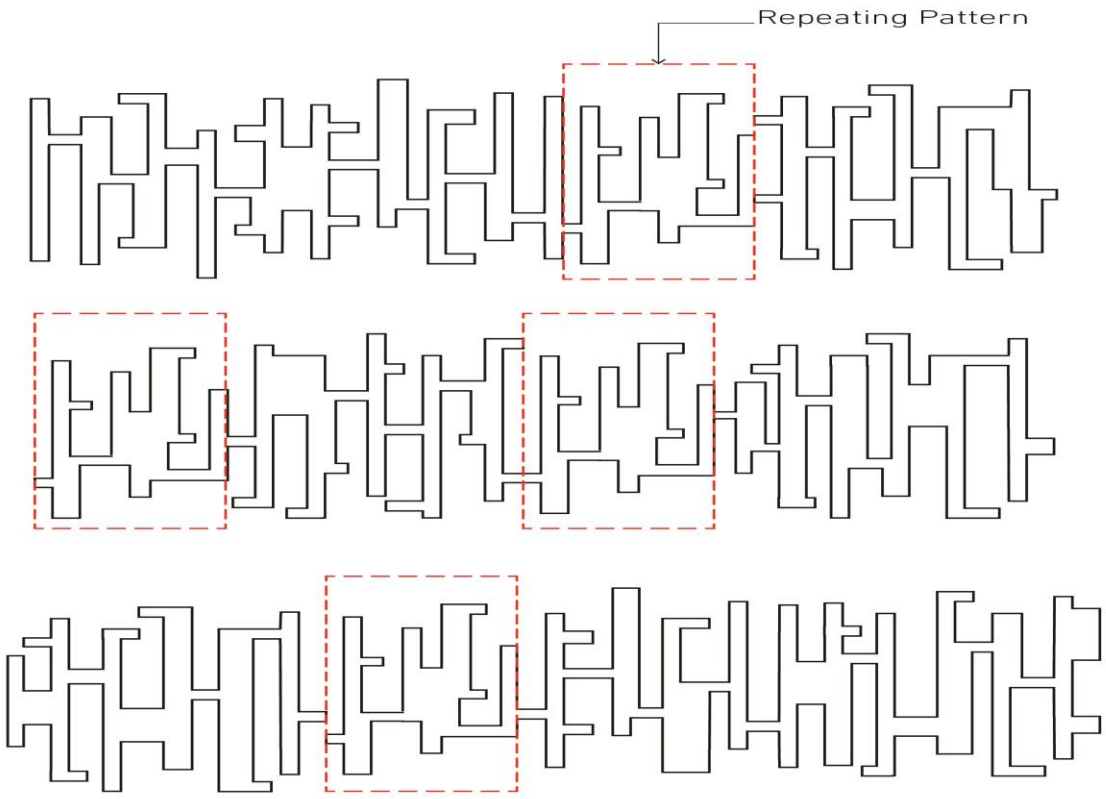


Fig 1.2 Simple example of Repeating Pattern in Layout

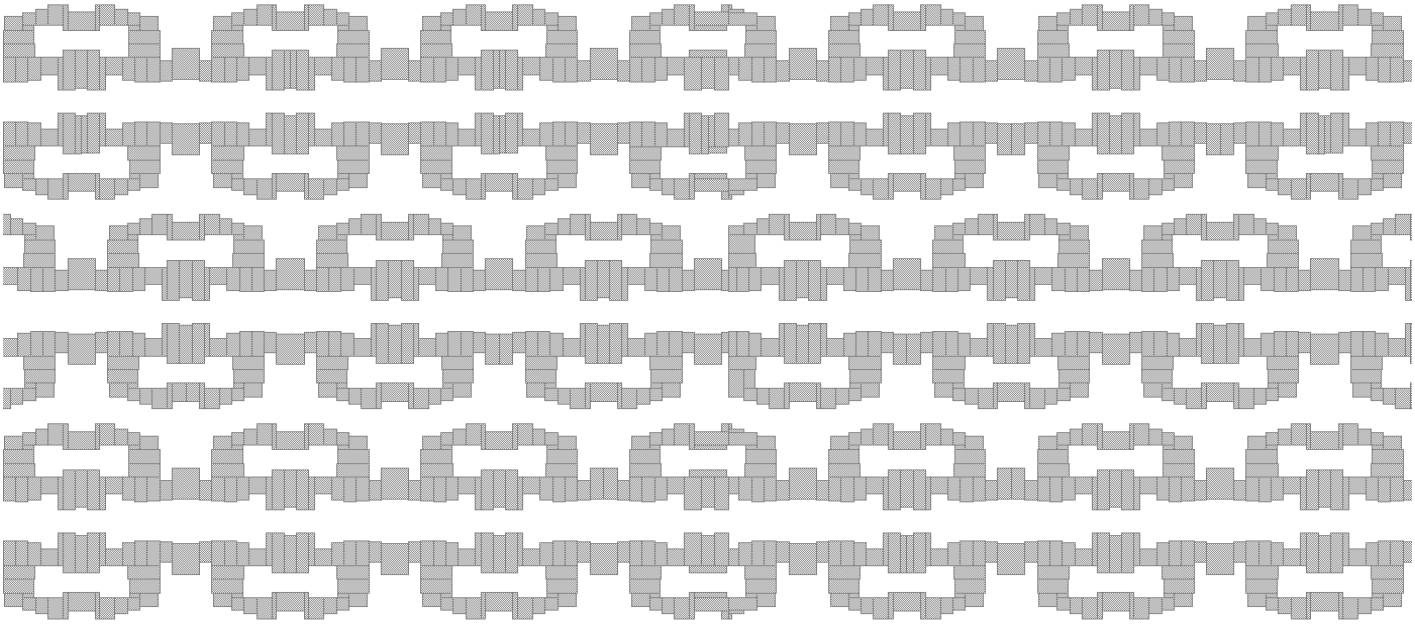


Fig 1.3 Millions of rectangles forming Input Layout

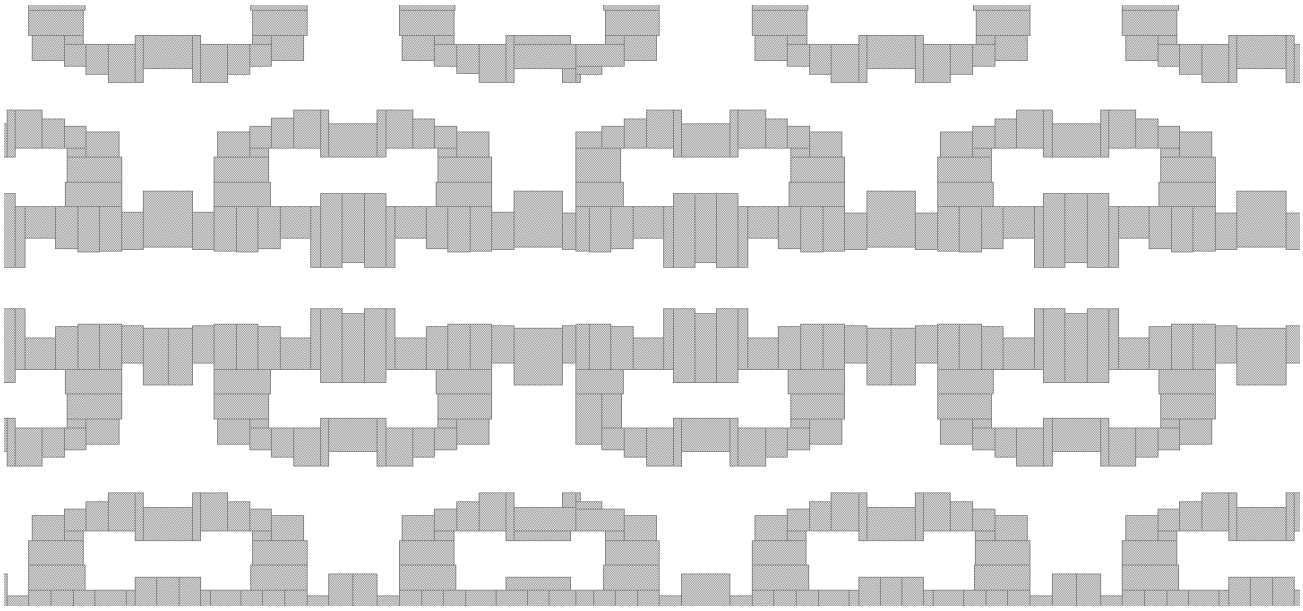


Fig 1.4 Types of Repeating Polygon in Input Layout

Chapter 2

Rectilinear Polygon Fracturing

2.1 Overview

After the physical mask layout is created for a circuit for a specific design process, the layout is measured by a set of geometric constraints, or rules, for that process. The main objective of design rule checking (DRC) is to achieve a high overall yield and reliability for the design. Once the mask data layout has been created and modified to accommodate various Resolution Enhancement Techniques (RET) algorithms, the final photo-mask still needs to be written. Usually, the data must be flattened to some degree, and the polygons must be reduced to a simple set of structures (typically rectangles and trapezoids) that the machine can use to write the patterns directly. This process of data conversion is called fracturing. **Polygon fracturing (partitioning)** converts the complex polygons generated by the layout process, into non-overlapping trapezoids suitable for mask writing.

- **Layout Data**: It is the representation of an IC in terms of geometrical shapes spread across one or more layers. Stored in file formats Oasis.

- **Mask Data**: The layout data is fractured into rectangles/trapezoids, required to synthesize photo-masks (stencil). It contains only rectangles and/or trapezoids.

Traditionally, after an IC design has been converted into a physical layout, the timing verified, and the polygons certified to be DRC-clean, the IC was ready for fabrication. The data files representing the various layers were shipped to a mask shop, which used mask-writing equipment to convert each data layer into a corresponding mask, and the masks were shipped to the fab where they were used to repeatedly manufacture the designs in silicon.

2.2 Mask data preparation (MDP)

It is the procedure of translating a file containing the intended set of polygons from an integrated circuit layout into set of instructions that a photo-mask writer can use to generate a physical mask. It usually involves mask fracturing where complex polygons are translated into simpler shapes, often rectangles and trapezoids that can be handled by the mask writing hardware. The partitioning run time and quality directly impacts the cost, integrity, and quality of the written mask. Factors determining **quality of mask data** may be multiple:

- Rectangle Count
- Figure Count & Shot Count
- Critical Dimension
- High Temperature

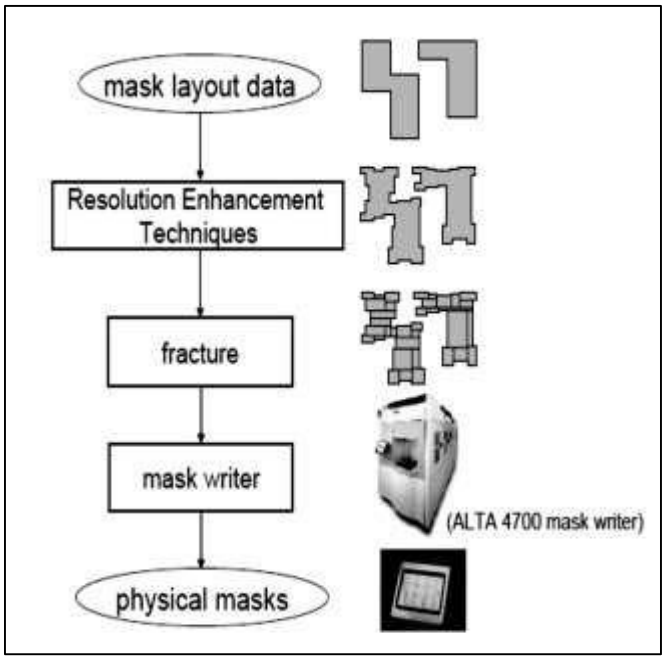


Fig 2.1 MDP Flow

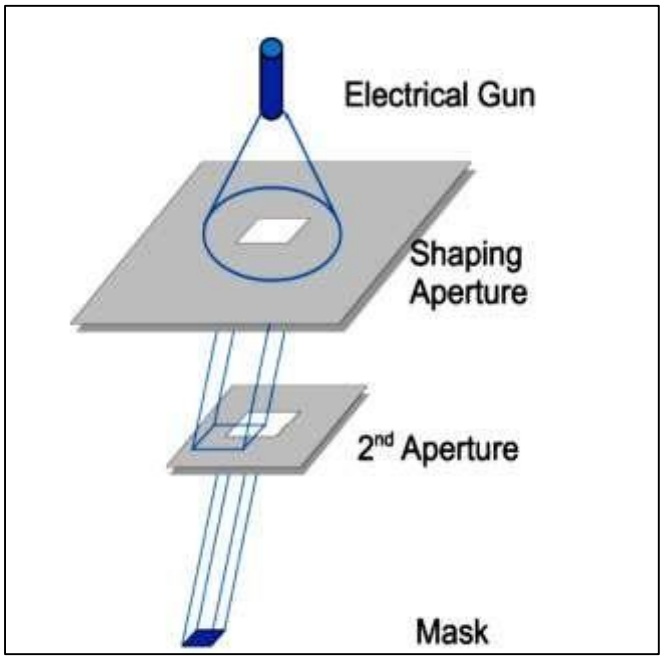


Fig 2.2 Conventional VSB writer

2.2.1 Design Rule Checking (DRC)

It is the area of Electronic Design Automation that determines whether the physical layout of a particular chip layout satisfies a series of recommended parameters called **Design Rules**. Design rule checking is a major step during Physical verification signoff on the design, which also involves LVS (Layout versus schematic) Check, XOR Checks, ERC (Electrical Rule Check) and Antenna Checks. Sometimes known as geometric verification, this involves verifying if the design can be reliably manufactured given current photolithography limitations. A **design rule set** specifies certain geometric and connectivity restrictions to ensure sufficient margins to account for variability in semiconductor manufacturing processes, so as to ensure that most of the parts work correctly. Typical design rule checks involve reporting the following stats and checking for violations:

- Small figure count.
- Split CD count.
- Total Figure count.
- Total Shot count.
- XOR check.

The **main objective** of design rule checking (DRC) is to achieve a high overall yield and reliability for the design. If design rules are violated the design may not be functional. To meet this goal of improving die yields, DRC has evolved from simple measurement and Boolean checks, to more involved rules that modify existing features, insert new features, and check the entire design for process limitations such as layer density. A completed layout consists not only of the geometric representation of the design, but also data that provides support for the manufacture of the design. While design rule checks do not validate that the design will operate correctly, they are constructed to verify that the structure meets the process constraints for a given design type and process technology. DRC software usually takes as input a layout in the GDSII standard format and a list of rules specific to the semiconductor process chosen for fabrication. From these it produces a report of design rule violations that the designer may or may not choose to correct.

The physical mask layout consists of shapes on drawn layers that are grouped into one or more cells. A cell may contain a placement of another cell. If the entire design is represented in one cell, it is a **flat design**; otherwise it is a **hierarchical design**.

DRC is a very computationally intense task. Usually DRC checks will be run on each sub-section of the Application Specific Integrated Chip (ASIC) to minimize the number of errors that are detected at the top level. If run on a single CPU, customers may have to wait up to a week to get the result of a Design Rule check for modern

designs. Most design companies require DRC to run in less than a day to achieve reasonable cycle times since the DRC will likely be run several times prior to design completion.

2.2.2 Standard Verification Rule Format (SVRF)

Language based DRC products define rules in a language to describe the operations needed to be performed in DRC. For example, Machine uses Standard Verification Rule Format (SVRF) language in their DRC rules files and Magma Design Automation is using **Tcl-based language**. A set of rules for a particular process is referred to as a run-set, rule deck, or just a deck.

The C MDP product line completes the integrated flow from IC design to IC mask manufacturing. The flow concludes with the output of the most important mask writer formats for advanced mask-making in the sub wavelength era, **Variable-Shaped-Beam (VSB)** formats and **OASIS**. As feature sizes continue to shrink while IC density increases, the volume of data required to describe an IC will continue to grow exponentially. To mitigate this, a new data format called Open Artwork System Interchange Standard (OASIS) was approved by SEMI recently.

Chapter 3

Fracturing Methods

3.1 Previous Work

Fracture of a polygon into basic shapes rectangles is a well-studied problem. The standard formulation is to minimize the number of shots subject to certain constraints. **Ohtzuki (1982)** has given an exact **O ($n^{5/2}$) algorithm** for polygon fracture into rectangles where **n** is the number of vertices of a polygon. The algorithm is based on finding a maximum independent set in a bipartite graph where vertices correspond to certain lines slicing the given polygon.

Imai and Asano (1986) have further sped up this algorithm to **O($n^{3/2}$ logn)** and also generalized it to the optimal partition into rectangles. Unfortunately, these theoretically nice algorithms but pattern Identification will speed-up this algorithm.

Nakao et al(2000) have developed a fairly complicated ad hoc heuristic based on the generalization of the same bipartite graph which takes in account all other constraints except the constraint. In fact, they have introduced a different objective – minimize the weighted length of slivers and slices cutting through critical fea-tures while minimizing shot number over all obtained solutions that are (sub)optimal with respect to the new objective. Their heuristic does not guarantee optimum fracture.

B. Yu, J.-R. Gao and D. Z. Pan (2013) proposed a L-Shape Based Layout Fracturing for E-Beam Li-thography. They proposed two novel algorithms. The first one, rectangular merging (RM), starts from a set of rectangular fractures and merges them optimally to form L-shape fracturing. The second algorithm, direct L-shape fracturing (DLF), directly and effectively fractures the input layouts into L-shapes with sliver minimization. The experimental results show that their algorithms are very effective

Chapter 4

Algorithms Used

4.1 Scan-Line Algorithm

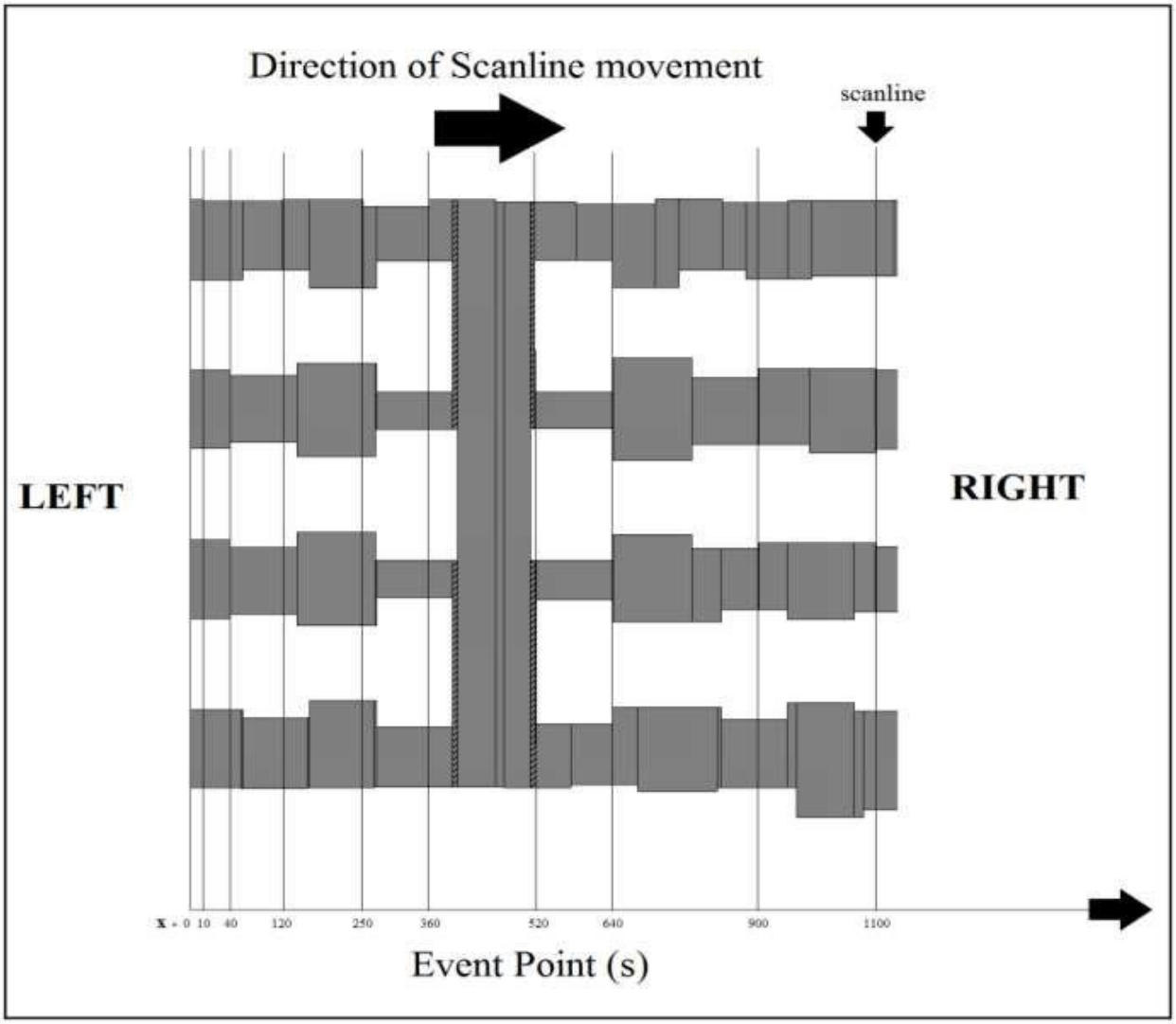


Fig 4.1 Components of an edge-based Scan-line Algorithm. The layout extent is swept from left to right, stopping at every event point.

Scan-line based sweep algorithms have become the predominant form of low-level geometric analysis. A scan-line sweep analyses relationships between objects that intersect a virtual line, either vertical or horizontal, as that line is swept across the layout extent

Scan Object: The rectangles (present in the layout extent) are provided as input to the scan line and are ordered first in order of increasing **X** (of their left edges) and then in order of increasing **Y** (of their bottom edges).

Event Point: An event point (as defined here) is every distinct **X** at which either a scan object is leaving the scan-line or is entering into it. The scan-line must stop at every event point and appropriately iterate over all its objects, constantly updating itself.

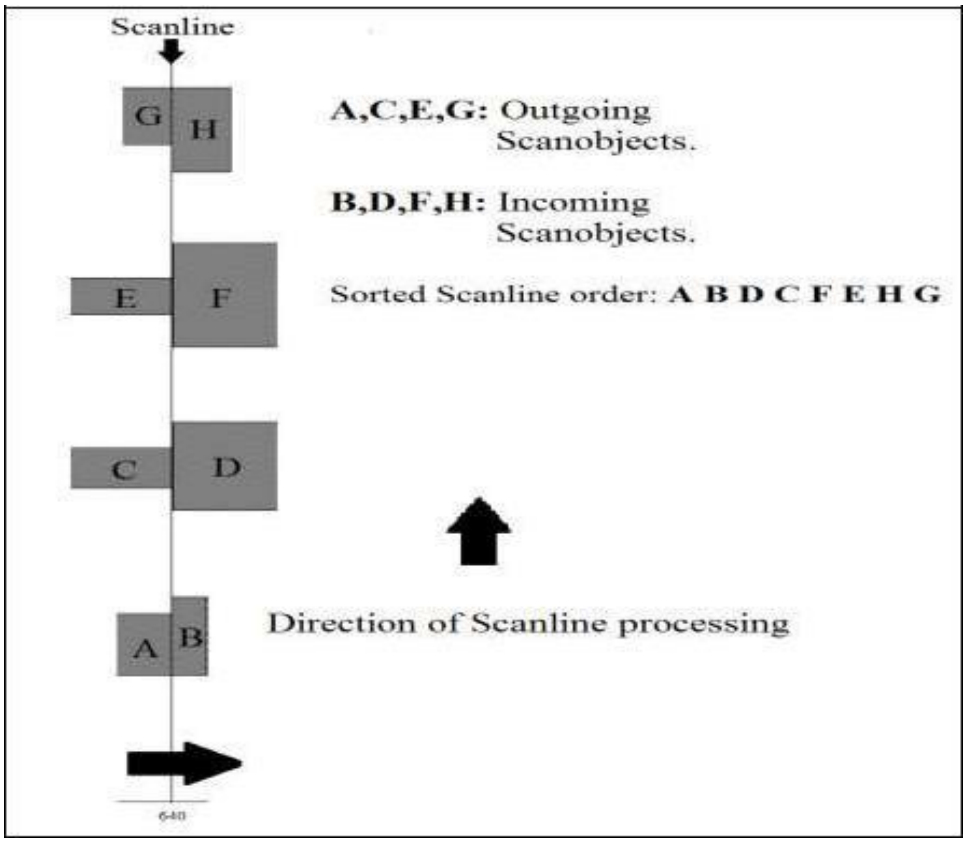
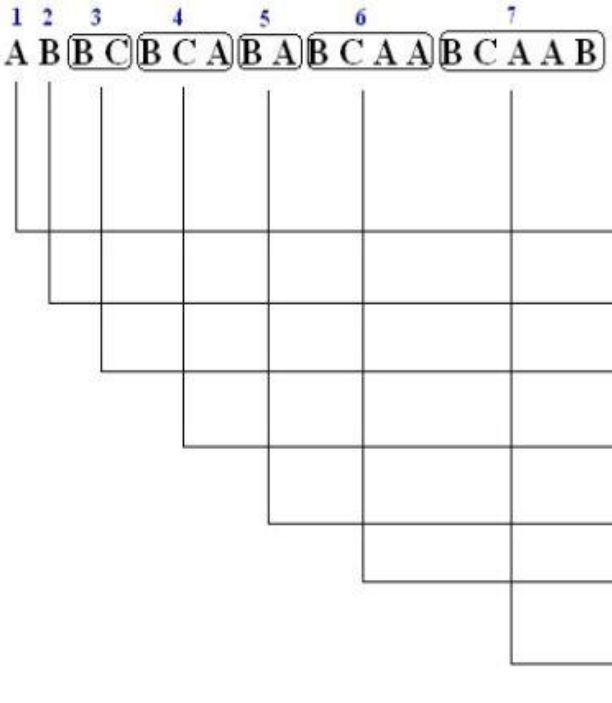


Fig 4.2 Status of the Scan-Line at event point

4.1 LZ Compression Algorithm

The LZ algorithm works by constructing a dictionary of substrings, which we will call "phrases," that have appeared in the text. The LZ algorithm constructs its dictionary on the fly, only going through the data once. This means that you don't have to receive the entire layout data before starting to encode it. The algorithm parses the sequence into distinct phrases. We do this greedily.

Example-1Encode (i.e. compress) the string **ABBCBCABABCAABCAAB** using the LZ78 algorithm.



	Dictionary	
output	index	string
(0, A)	1	A
(0, B)	2	B
(2, C)	3	BC
(3, A)	4	BCA
(2, A)	5	BA
(4, A)	6	BCAA
(6, B)	7	BCAAB

Fig 4.3 Dictionary formation during Compression

The compressed message is: **(0,A)(0,B)(2,C)(3,A)(2,A)(4,A)(6,B)**

Advantage of compression Algorithm:-

Example: Uncompressed String: **ABBCBCABABCAABCAAB**

Number of bits = Total number of characters * 8

= 18 * 8

= 144 bits

Suppose the code-words are indexed starting from 1:

Compressed string (code-words): **(0, A) (0, B) (2, C) (3, A) (2, A) (4, A) (6, B)**

Code-word index	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Each code word consists of an integer and a character:

The character is represented by **8** bits. The number of bits **n** required to represent the integer part of the code-word with index **i** is given by:

$$n = \begin{cases} 1 & \text{if } i = 1 \\ \lceil \log_2 i \rceil & \text{if } i > 1 \end{cases}$$

Alternatively number of bits required to represent the integer part of the code-word with index **i** is the number of significant bits required to represent the integer **i – 1** .

Code-word	**(0, A)**	**(0, B)**	**(2, C)**	**(3, A)**	**(2, A)**	**(4, A)**	**(6, B)**
index	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Bits: (1 + 8) + (1 + 8) + (2 + 8) + (2 + 8) + (3 + 8) + (3 + 8) + (3 + 8) = 71bits

The actual compressed message is: **0A0B10C11A010A100A110B**

Pseudo Algorithm

START

```
Dictionary ← empty ; Prefix ← empty ; DictionaryIndex  ← 1;
while(characterStream is not empty){
        Char ← next character in characterStream;
    if(Prefix + Char exists in the Dictionary)
        Prefix ← Prefix + Char ;
     else {
         if(Prefix is empty)
             CodeWordForPrefix  ← 0 ;
         else
             CodeWordForPrefix ← DictionaryIndex for Prefix ;
         Output: (CodeWordForPrefix, Char) ;
             insertInDictionary( ( DictionaryIndex , Prefix + Char) );
             DictionaryIndex++ ;
             Prefix ← empty ;
         }
}
if(Prefix is not empty){
        CodeWordForPrefix ← DictionaryIndex for Prefix;
   Output: (CodeWordForPrefix,   ) ;
}
```

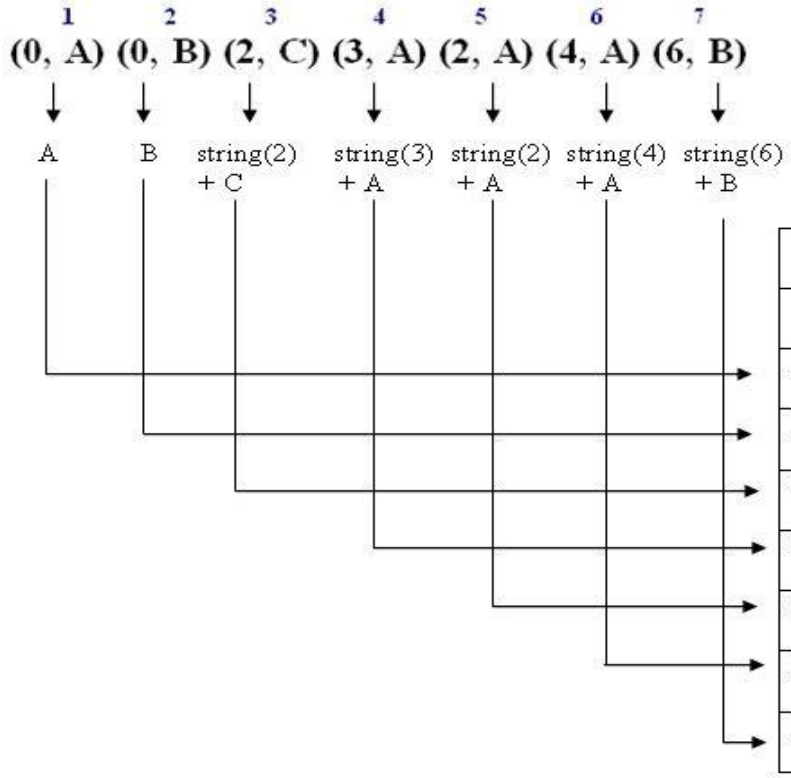
END

4.2 LZ Decompression Algorithm

The Decompression process for LZ is also very simple. In addition, it has an edge over static compression methods because no dictionary or other pre-existing information is necessary for the decoding algorithm–a dictionary identical to the one created during compression is re-built during the process. Both encoding and decoding programs must start with same initial dictionary. Here's how it works The LZ decoder first reads in an index, looks up the index in the dictionary, and returns the substring associated with the index. The first character of this substring is appended to the current working string. This new concatenation is added to the dictionary .The decoded string then becomes the current working string (the current index, i.e. the substring, is remembered), and the process repeats.

Example 1

Decompress the sequence (0, A) (0, B) (2, C) (3, A) (2, A) (4, A) (6, B)



	Dictionary	
output	index	string
A	1	A
B	2	B
BC	3	BC
BCA	4	BCA
BA	5	BA
BCAA	6	BCAA
BCAAB	7	BCAAB

Fig 4.4 Dictionary formation during Decompression

The decompressed String is: **ABBCBCABABCAABCAAB**

Pseudo Algorithm

START

```
Dictionary ← empty ; DictionaryIndex ← 1 ;
while(there are more (CodeWord, Char) pairs in codestream){
        CodeWord ← next CodeWord in codestream ;
        Char ← character corresponding to CodeWord ;
        if(CodeWord = = 0)
         String ← empty ;
        else
           String ← string at index CodeWord in Dictionary ;
        Output: String + Char ;
        insertInDictionary( (DictionaryIndex ,  String + Char) ) ;
    DictionaryIndex++;
}
```

END

Chapter 5

Experimental Walkthrough

5.1 Hardware Specifications

Major test runs of the algorithm were done on a x86 architecture 64 bit-Linux Virtual Machine having processor specification as: **Intel(R) Core(TM) i7 CPU E5-2680 v3 @ 2.50 GHz, 16 GB RAM**. For slightly bigger test cases special VMs' with at-least 256 GB RAM support, allotted on Calibre Grid infrastructure were employed. The algorithm is implemented as an independent module in **C++** and is compiled along with Calibre MDP flow. While running, the function API is called from Calibre Flow, which returns the appropriate output to the flow after processing.

5.2 Data Description

The input OASIS files were ASIC layouts containing only Manhattan polygons, and were as large as 100 mm layouts. Input of polygon will be given by the horizontal edges with its direction and end point of edges with its polygon number.

5.3 Approach Explained

Approach is divided in three part and output of one part will become the output of next part-

A1. First Part is Converting Non Monotone Polygons to Monotone Polygons (using Scan-Line approach)

Starting Approach –

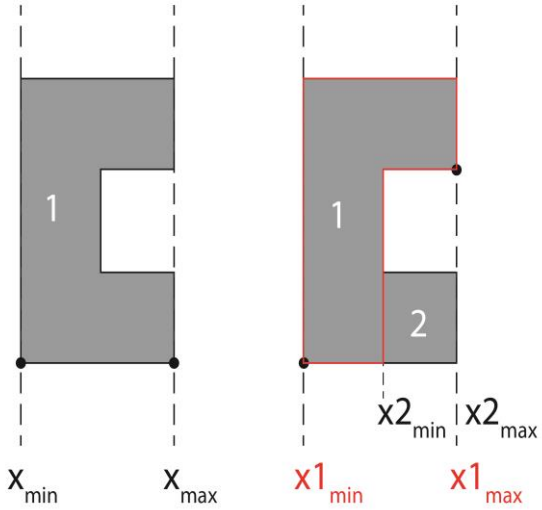
In 1st Iteration divide the Polygons into rectangles and give same Polygon number to the rectangles which are overlapping. Then in 2nd Iteration merge the rectangles which have same polygon number.

Final Approach/ Implemented Algorithm –

Instead of breaking the polygons in rectangle and then merging the rectangles, Divide the edge of polygon in two edges and assign different polygon number to both edges if the rectangles are not overlapping(i.e. Break the edges only if rectangles are not overlapping).

Advantage of second approach is we don't have to iterate through input data twice. Time complexity will be reduced to half of previous approach.

Monotonicity of polygon



Xmin, Xmax is dividing polygon in two chains.

If both the chains are increasing or decreasing then it is monotone else it is non-monotone polygon

If not monotone divide polygon in rectangles and merge the rectangles which are overlapping to make it monotone

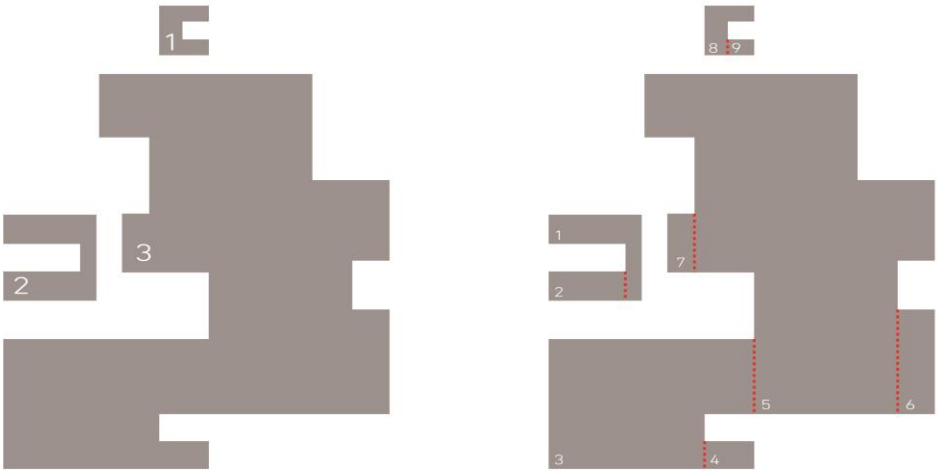


Fig 5.1 Simple Non-Monotone Polygon and Monotone Polygon

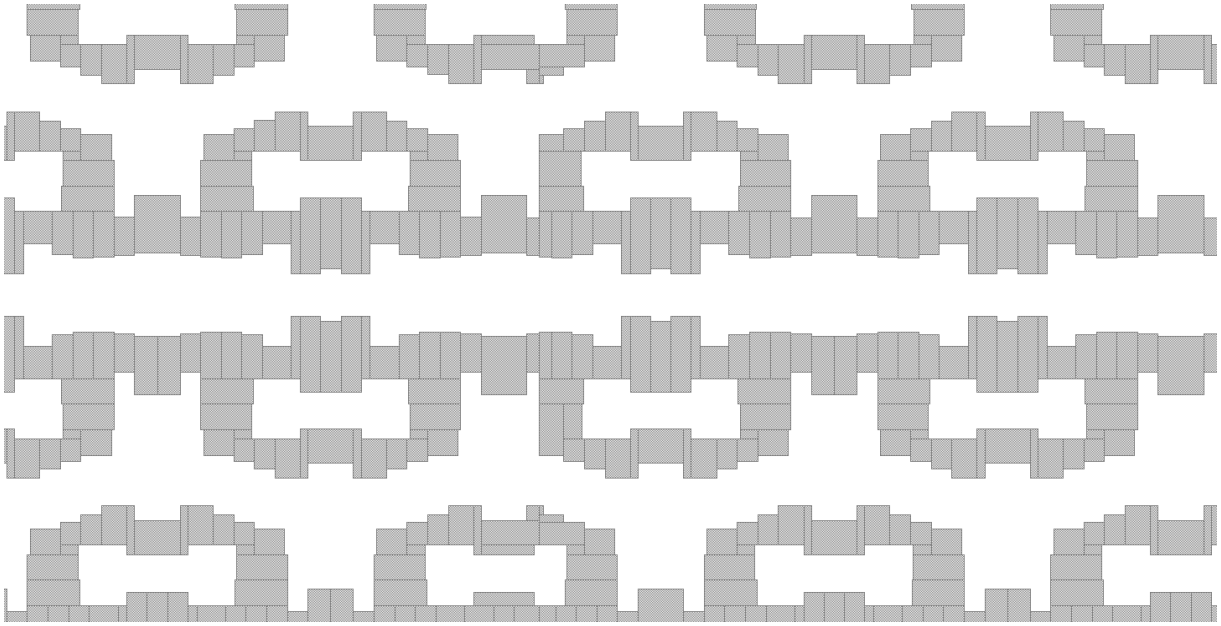


Fig 5.2 Non-Monotone Polygon

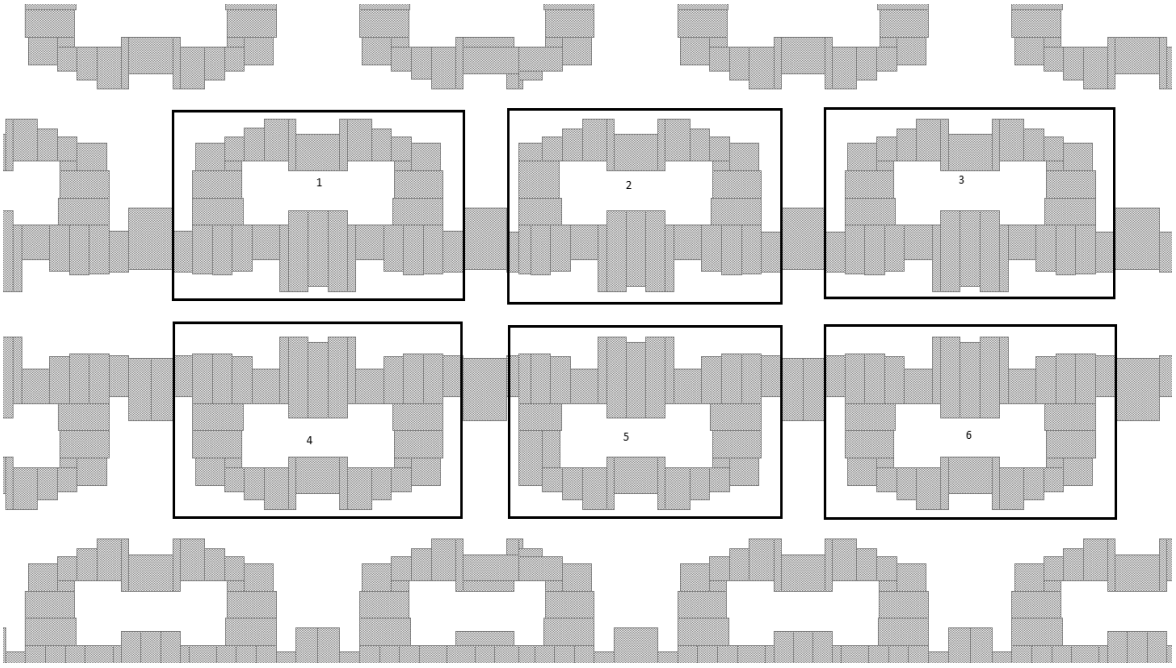


Fig 5.3 Repeating Pattern Marked

Fig 5.2 shows the Input layout of polygon which are not monotone. Fig 5.3 shows the repeating pattern in the Input Layout.

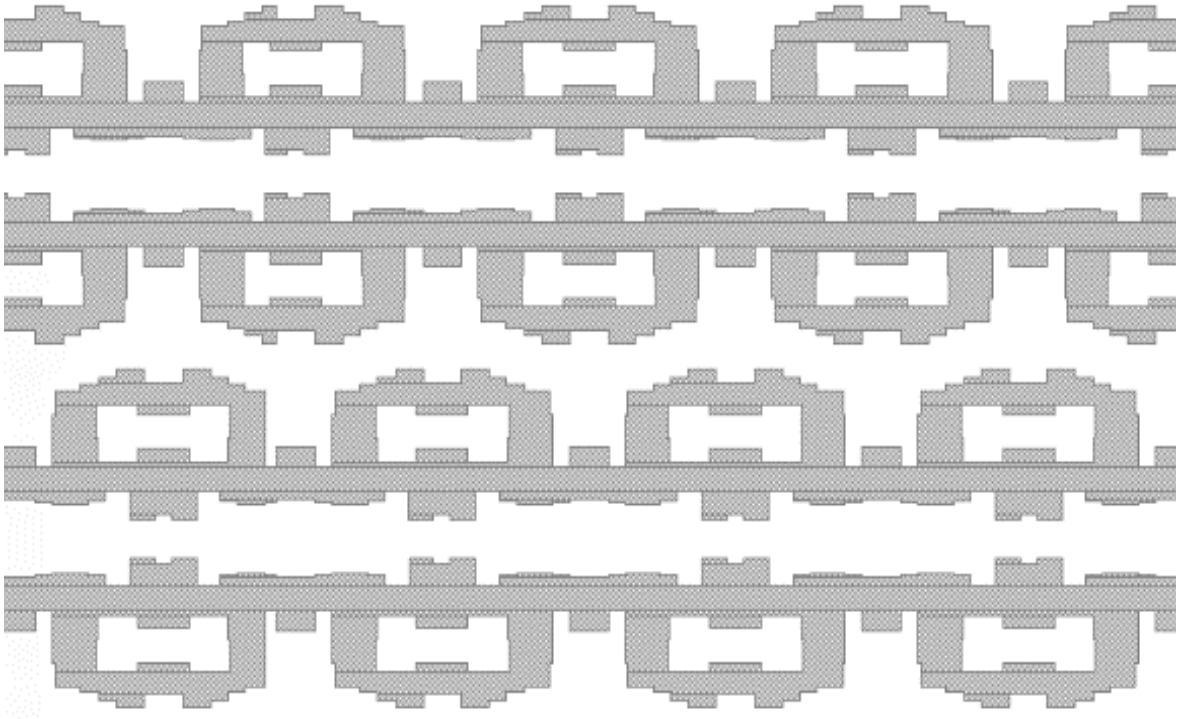


Fig 5.4 Monotone Polygon Layout

A2. Mapping of rectangles and generation of string

(When using 16 bit):-

[For 16 bit mapping, 8 bit will be used for representing length and other 8 bit will be used for representing breadth]

For mapping the length/ breadth of rectangle to 8 bit (decimal to binary). I am using Bit-Masking technique (bitwise "&" operator).

INPUT - Length or breadth of rectangle in decimal

1. Initialize ‘mask’ variable with 128 [10000000] (*for 16 bit only, mask value change according to mapping)*

2. Perform AND of two number’s (i.e. length/breadth and Mask)

3. Check whether the Result of AND is 0 or not, if Yes output bit is 0 otherwise 1

4. Right shift mask variable by 1 [0100 0000]

5. Now check for Second bit, whether it is 0 or 1

6. Go to step 3 until ‘mask’ becomes Zero [00000000].

OUTPUT- Length or breadth of rectangle in 8 bit

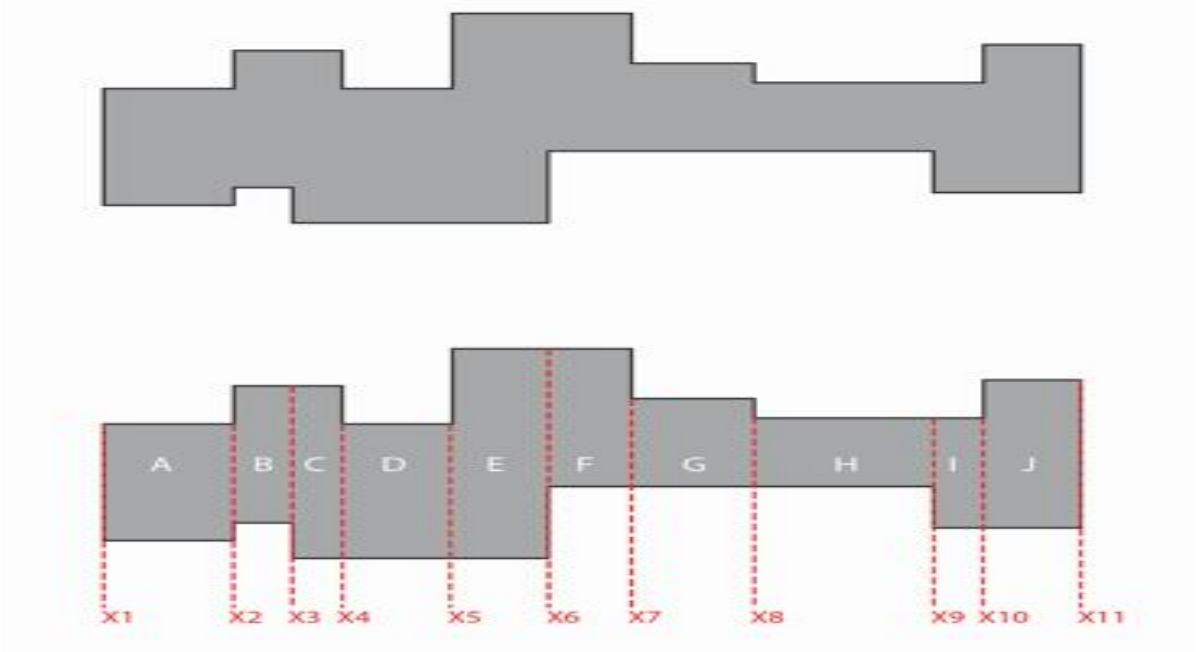


Fig 5.5 Monotone polygon divided in rectangle for mapping

For mapping 32 bit and 8 bit we will use same algorithm, we just have to change the mask value.

(When using 32 bit mapping) Mask value will be 32768 [1000 0000 0000 0000]

16 bit will be used for representing length and 16 bit will be used for representing breadth].

(When using 8 bit mapping) Mask value will be 15 [1000].

4 bit will be used for representing length and 4 bit will be used for representing breadth].

Advantage of using Bit masking is we don't have to change the whole algorithm for different mapping (i.e. for 32, 16 and 8) we just have to change the mask value.

Difficulties while mapping:-

First difficulty while mapping the rectangle in unique character is, can't Map all the rectangles because some rectangles are very big whose dimension exceed 255*255. 1 byte is very small (i.e. dimension till 255*255 rectangle only). So idea is to use more byte 2 byte or 4 byte i.e. (16 bit or 32 bit). If rectangle dimension is still bigger than leave that rectangle.

For 8 bit Mapping:-

1 rectangle is mapped to 8 bit unique char, can map from (0 X 0) to (15 X 15)

Length X Breadth = 8 bit Unique Char

(4 bit) (4bit) = 8 bit

For 16 bit Mapping:-

1 rectangle is mapped to 16 bit unique char, can map from (0 X 0) to (255 X 255)

Length X Breadth = 16 bit Unique Char

(8 bit) (8bit) = 16 bit

For 32 bit Mapping:-

1 rectangle is mapped to 32 bit unique char, can map from till (65535 X 65535)

Length X Breadth = 32 bit Unique Char

(16 bit) (16bit) = 32 bit

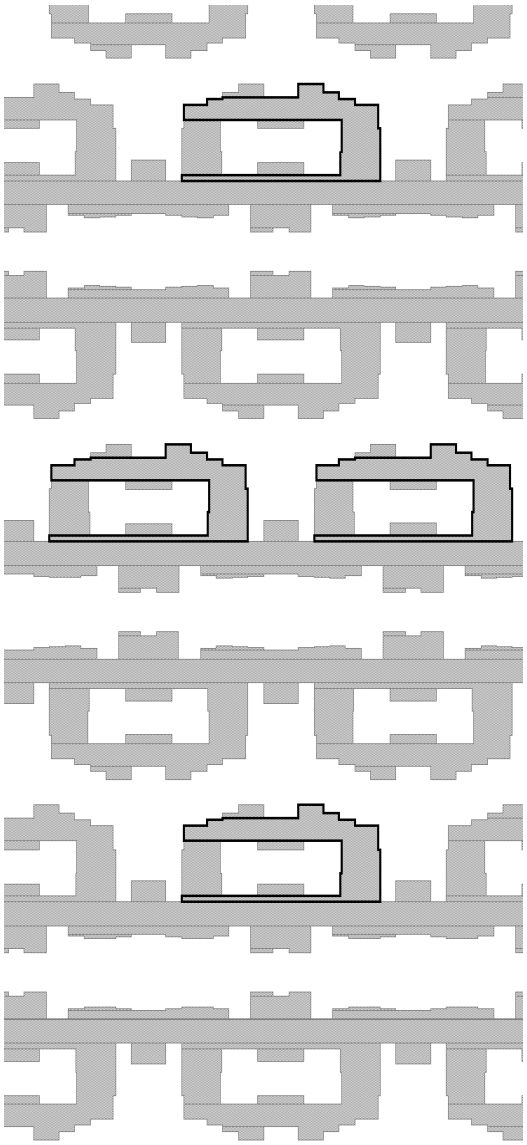


Fig 5.6(a)

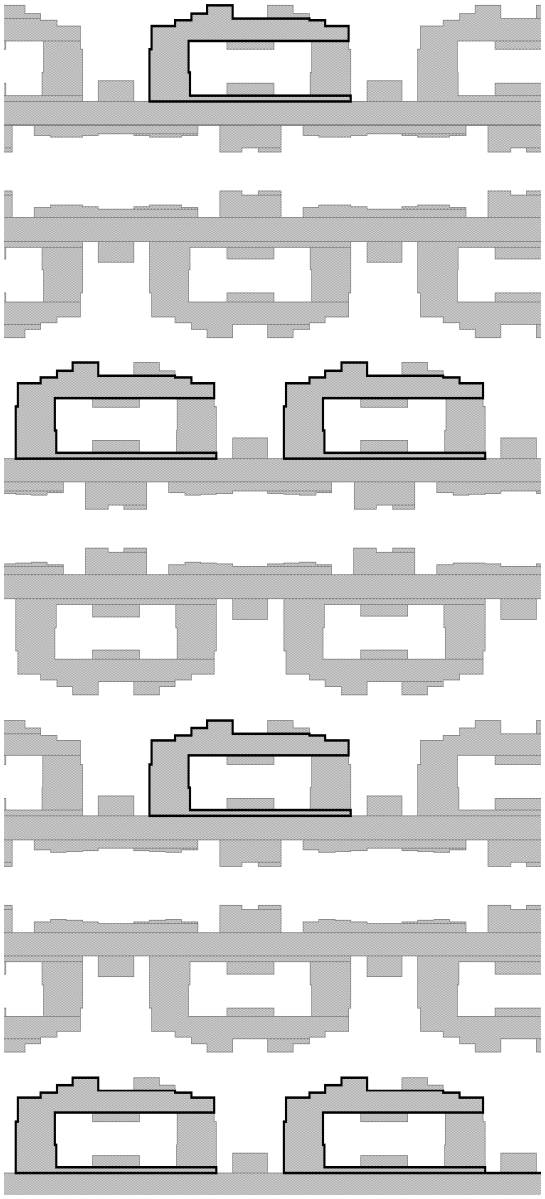


Fig 5.6(b)

Fig 5.6(a) and 5.6(b) shows two different types of repeating pattern in polygon layout. These repeating pattern is Monotone polygon where one monotone represents on substring two monotone polygons which are same in shape but different orientation will have different unique character,

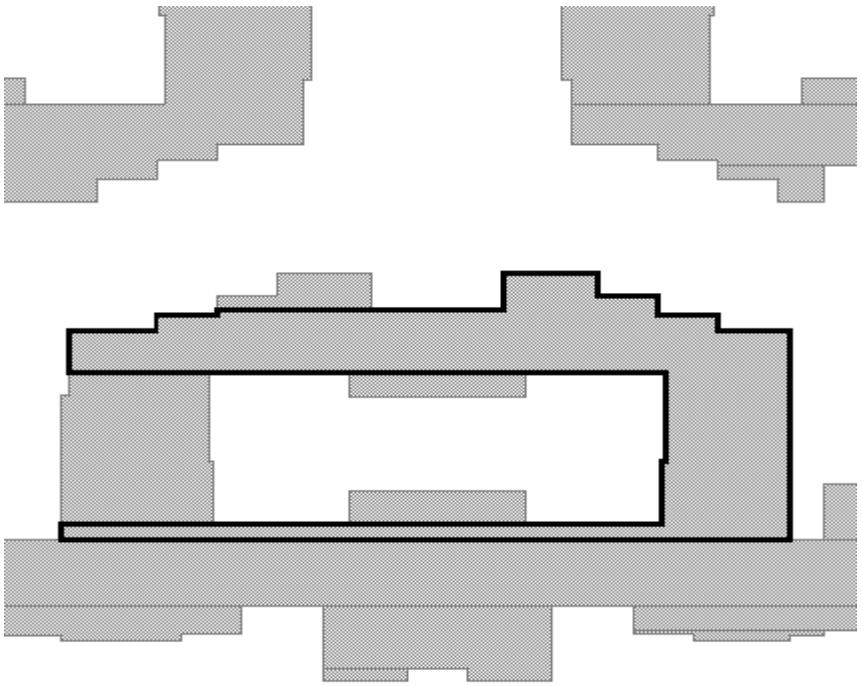


Fig 5.7(a)

Fig 5.7(a) shows one repeating monotone pattern which represents one substring and these type of substring will be divided by a separator “#”. Fig 5.7(b) shows how a substring will be divided in rectangle one rectangle here is converted in either 16 bit, 8 bit or 32 bit and sequence of this forms a substring.

Length and breadth for 8 bit mapping will be divided in 4-4 bit. Similarly, for 16 and 32 bit length and breadth will be divided in 8-8 and 16-16 bit respectively. So by using 32 bit we can map more rectangle in comparison to 16 bit and 8 bit.

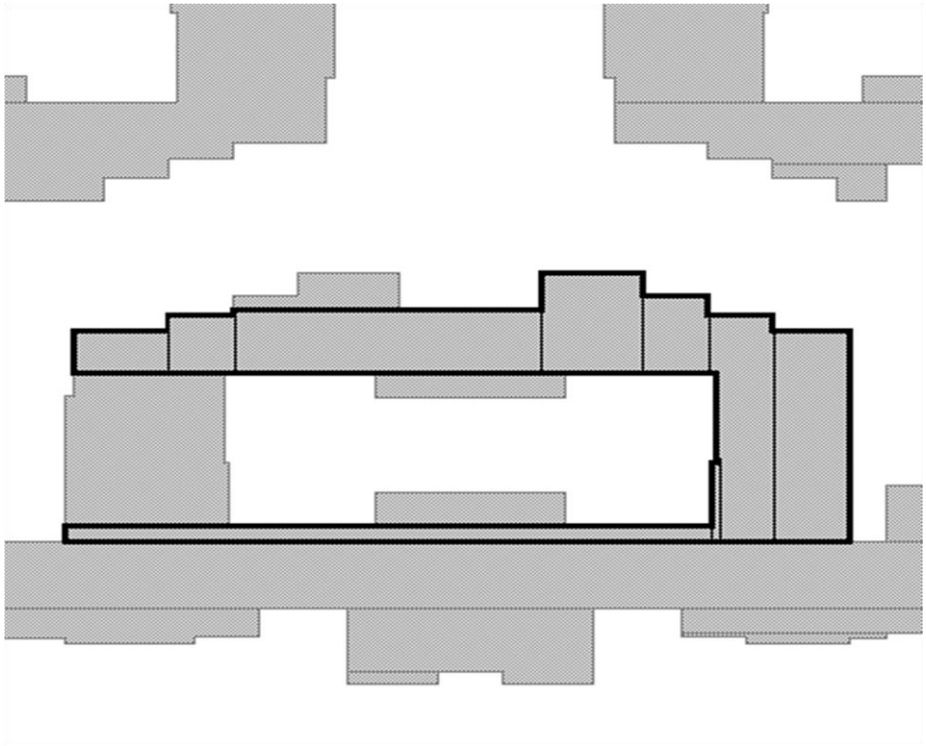


Fig 5.7(b)

A3. Third and last part gives the data for Number of repetition of particular substring-pattern)

Input - Substring of 16 bit characters separated by "# ".

Using LZ Compression this part is done in three step.

Step 1- Iterate through input data and store the repeating pattern in dictionary using compression algorithm. This give output of partially compressed string

Step 2- Iterate through dictionary and remove the substring which is not matching the minimum repetition criteria.(**Important** – This step removes the pattern which is not repeating certain no. of time .Not to get confused with removal of pattern which have less number of element/characters) because removal of minimum number of element was not giving the output we needed.

Step 3- Iterating through partially compressed string so that we can compress it properly and this time also we are using LZ compression but we are not adding anything new to dictionary.

Final Output – Substring pattern with its number of repetition.

5.4 Experimental Milestones

Given an input of Manhattan Polygon Layout Data.

M1.Conversion of Non- Monotone Polygon to Monotone Polygon using Scan-Line approach

M2. First we will break the Polygon in Rectangle of size (255*255) because we want to represent every rectangle with unique 16 bit so that we can convert rectangle (2D) to 16 bit char (1D) so that we can give input to LZ Compressor.

M3. After breaking the rectangle we get rectangle of max size (255*255) which is to be mapped in 16 bit unique char.

Example- 0 * 1 is mapped to 00000000+00000001

24 * 2 is mapped to 00011000+00000010

2 * 24 is mapped to 00000010+00011000

.

.

100 * 100 is mapped to 01100100 + 01100100

.

.

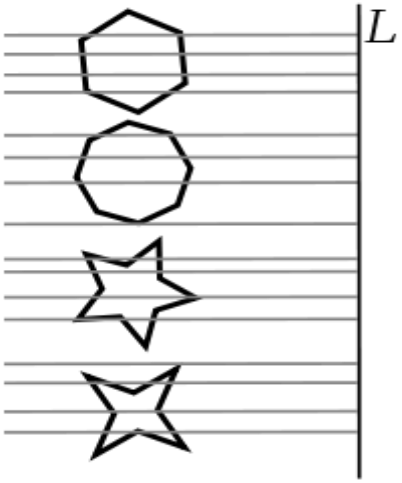
254 * 254 is mapped to 11111110 +11111110

255 * 255 is mapped to 11111111 +11111111

Similarly till 255* 255 just add the first 8 bit of length with 8 bit of breadth and we will get 16 bit unique char.

M4. So after this for making substring of these char we will use Scan-line approach and convert the polygon to monotone polygon. So making polygon monotone will give us a string and we insert “#” character as a separator in between polygon.

In geometry, a **polygon** P in the plane is called **monotone** with respect to a straight line L, if every line orthogonal to L intersects P at most twice.
Similarly, a polygonal chain C is called **monotone** with respect to a straight line L, if every line orthogonal to L intersects C at most once.

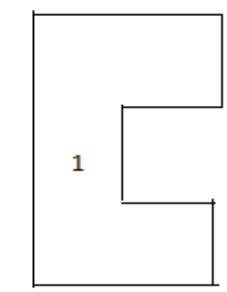


So our input String for LZ compressor will look like this-

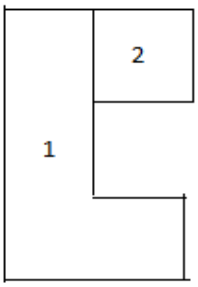
10001000,01010101,00000001,11111011,00100100,00001000#010100001,
01010101,00000001,11111011#00100100,00001000,00000001,11111011,00100100#
10001000,01010101,01010101,00000001,11111011,00100100,00001000,00001000#
010100001#00000001,11111011,00100100#00100100,00001000,00000001,1111101
1,00100100#10001000,01010101,01010101,00000001,11111011,00100100,0000100
0,00001000#010100001#00000001,11111011,00100100#00100100,00001000,00000
001,11111011,00100100#10001000,01010101,01010101,00000001,11111011,00100
100,00001000,00001000#010100001#00000001,11111011,00100100#00100100,000
01000,00000001,11111011,00100100#10001000,01010101,01010101,00000001,111
11011,00100100,00001000,00001000#010100001#00000001,11111011,00100100#0
0100100,00001000,00000001,11111011,00100100#10001000,01010101,01010101,0
0000001,11111011,00100100,00001000,00001000#010100001#00000001,11111011,
00100100#00100100,00001000,00000001,11111011,00100100#10001000,01010101,
01010101,00000001,11111011,00100100,00001000,00001000#010100001#0000000
1,11111011,00100100#00100100,00001000,00000001,11111011,00100100#1000100
0,01010101,01010101,00000001,11111011,00100100,00001000,00001000#

Where 16 bit is one character (i.e. one rectangle) and “#” is separator between different polygons.

M5. By making polygon monotone we are making polygon uniform as we will see there is either increasing or decreasing line when we move toward particular axis.



Not- Monotone



Monotone

In left polygon is not monotone but if we divide same polygon in two part 1 and 2 we will get both the polygon monotone.

M6. So when String we got at step 4 when given to compression algorithm it compress the string the string by changing the whole repeating pattern with the index of matching pattern in dictionary. So the output of compression algorithm we get a compressed string of uniform Manhattan polygon.

M7. Now after the fracture of given string we decompress the output string to get the original string which can be mapped back to rectangles and we will get the output layout.

Chapter 6

Results & Discussions

In this chapter, we will discuss about the results we get after running each algorithm and output obtained from each part of approach applied. Fig 6.1 shows the input layout of polygon (Not-Monotone).

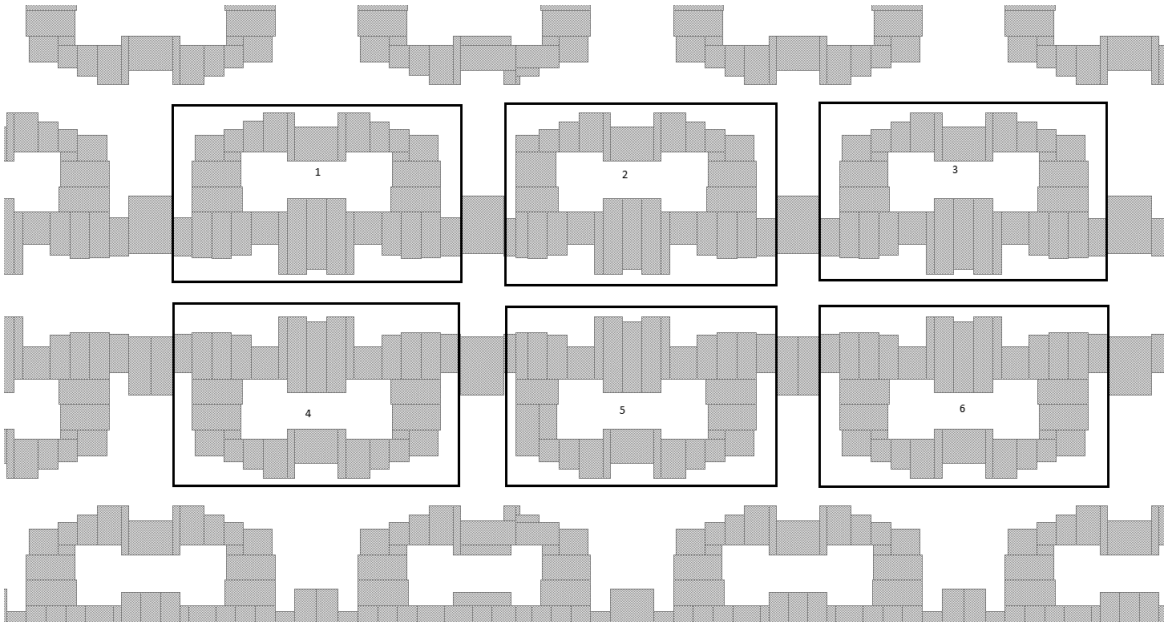


Fig 6.1 Input Layout of Polygon Data

After running the first algorithm on Input data we will get the monotone polygon which we is shown in fig 6.2. Repeating Monotone polygons are marked and they form a whole substring which is divided in rectangles and mapped in unique character and a string is formed which is separated by “#”.

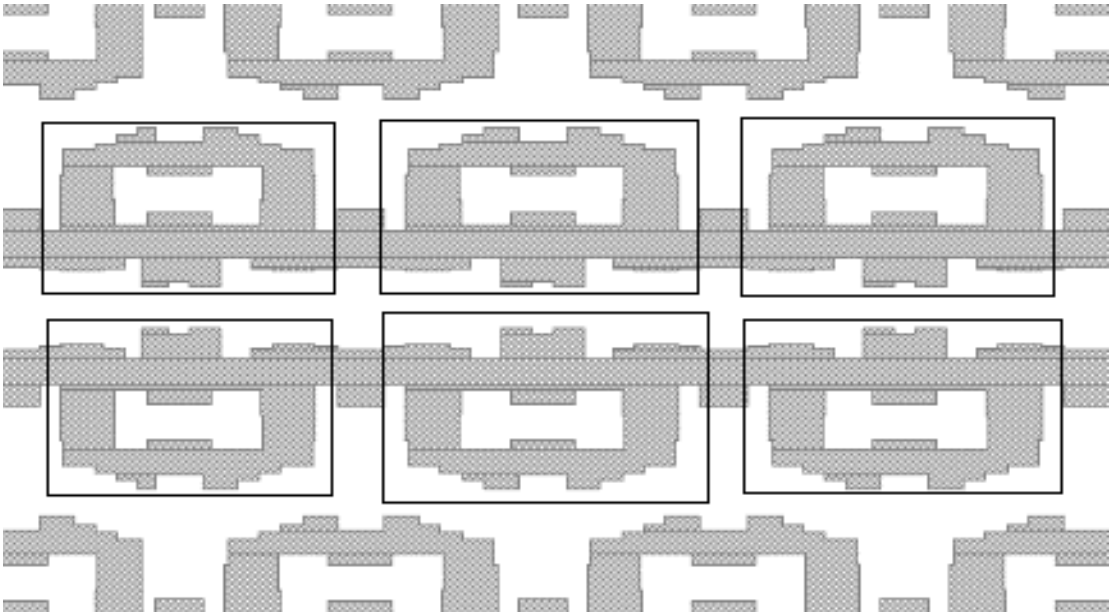


Fig 6.2 Monotone Output after using Scan-Line Approach

Now, we evaluate the performance of our mapping algorithms using the 3 different types of mapping bit.

Compared the Final output for file Uniformity-large while using 3 different types of mapping 32 bit, 16 bit and 8 bit.

Results are as expected - for 8 bit we are able to map less rectangle and as the mapping size increases we can map more and more no. of rectangles.

When we are using 8 bit mapping, number of rectangles mapped is very less compared to what we are getting in 16 bit mapping so it is better to neglect 8 bit mapping because it will not identify the repeating pattern**. (8 bit Neglected)**

Comparison between 16 bit and 32 bit is what need to be observed because result from either 16 bit or from 32 bit are going to be used depending on the result data obtained. It would be better if we can map more number of rectangle while increasing the time complexity bit more rather than mapping less rectangles for decreasing the time taken during compression of dictionary formation.

Comparison between 32 bit, 16 bit and 8 bit:-

COMPARISON 1			
	32 bit Mapping	16 bit Mapping	8 bit Mapping
Time Taken (in sec)	66	60	1
Size of Input (no. of bits + separator)	74,80,124	37,49,804	4,48,892
No. of separator in input (using only "#")	47,996	47,996	21,300
Total no. of Rectangles	2,32,254	2,31,363	53,449
Total no. of substring formed	1,294	1,205	66
No. of substring repeating less than 50 t	510	466	15

Table 6.1 Comparison of test-file Uniformity_large

When we use 32 bit mapping we can map **$4.3*10^9$** different rectangles in unique 32 bit characters.

When we use 16 bit mapping we can map **$6.5*10^4$** different rectangles in unique 16 bit characters.

As we can see by doubling the mapping size we can map approx. (10^5 times) more rectangles and this is what we need.

In Comparison_1 we can see for 32 bit mapping time taken is not much when we compare this to time taken for 16 bit mapping (only 6 sec increment).

But it is interesting to note that the number of rectangles in 32 bit is not much different from what we are getting in 16 bit mapping (only 900 more rectangles). (The reason of getting very less increment in number of rectangles is may be because the dimension of rectangles in file Uniformity-large is mostly under 255 *255 so we are covering most of rectangles in 16 bit only and nothing much left to map for 32 bit).

COMPARISON-2			
	32 bit Mapping	16 bit Mapping	8 bit Mapping
Time Taken (in sec)	112.1	75.74	0.81
Size of Input (no. of bits + separator)	53,18,556	24,36,476	2,51,967
No. of separator in input (using only "#")	30,716	30,716	13,887
Total no. of Rectangles	1,65,245	1,50,360	29,760
Total no. of substring formed	2,300	1,743	90
No. of substring repeating less than 50	985	660	20

Table 6.2 Comparison of test-file Uniformity_large_M2

Comparison for test-file Uniformity-large-M2 .Layout and shape of rectangles of this test file is similar to previous test file uniformity-large but this is magnified by 2.

When running compression algorithm on Uniformity-large-M2, the number of rectangles in this case is less than Uniformity-large as the size of rectangles are magnified and still the 8 bit mapping is not going to work because substring formed is very less.

For 16 and 32 bit, although the input size is decreased in Uniformity-large-M2.oas time taken by compression algorithm is bit more because number of substring formed in Uniformity-large-M2.oas is more than the substring formed in Uniformity-large.

Uniformity-Large-M2 VS Uniformity-Large				
	For 32 bit mapping		**For 16 bit mapping**	
	Uniformity-M2.oas	Uniformity.oas	**Uniformity-M2.oas**	Uniformity.oas
Time Taken (in sec)	112.1	66	75.74	60
Size of Input	53,18,556	74,80,124	24,36,476	37,49,804
No. of separator in input	30,716	47,996	30,716	47,996
Total no. of Rectangles	1,65,245	2,32,254	1,50,360	2,31,363
Total no. of substring formed	2,300	1,294	1,743	1,205
Substring repeating < 50 times	985	510	660	466

Table 6.3 Comparison between test-file Uniformity_large and Uniformity_large_M2

We can say that Time complexity of Compression algorithm is directly proportional to the no. of substring formed (i.e. Size of dictionary) as no. of pattern comparison in dictionary increases.

We can see the results of Comparison in Comparison_3.

Comparison between 16 bit and 32 bit for Unifromity_large_m2:-

Time taken for 32 bit mapping is 1.5 times the time taken by 16 bit mapping. As the dimension size is magnified, total no. of rectangle in 32 bit mapping is (15,000 more) than the total number of rectangle in 16 bit mapping.

32 bit mapping is taking more time than 16 bit mapping because no. of substring formed in 32 bit mapping is 1.35 times more than in 16 bit mapping.

Time taken by 16 bit and 32 bit for Uniformity-large was approximately same because in Uniformity-large the number of substring formed was also approximately same.

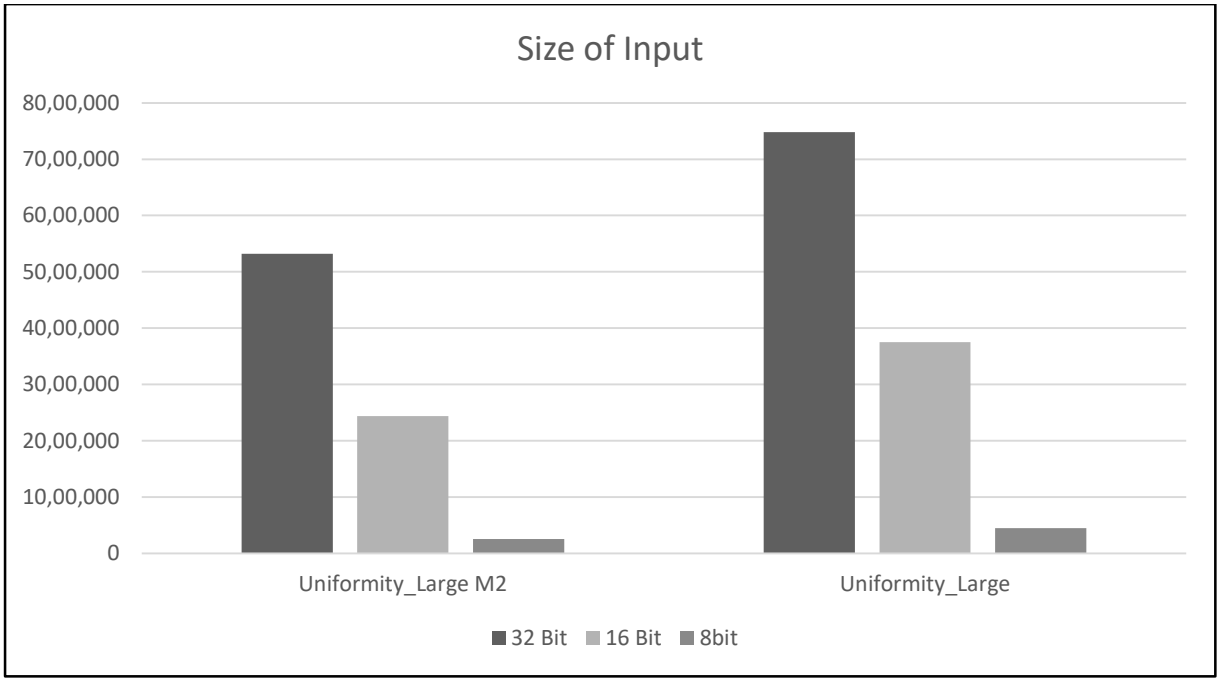


Fig 6.3(a)

Fig 6.3(b)

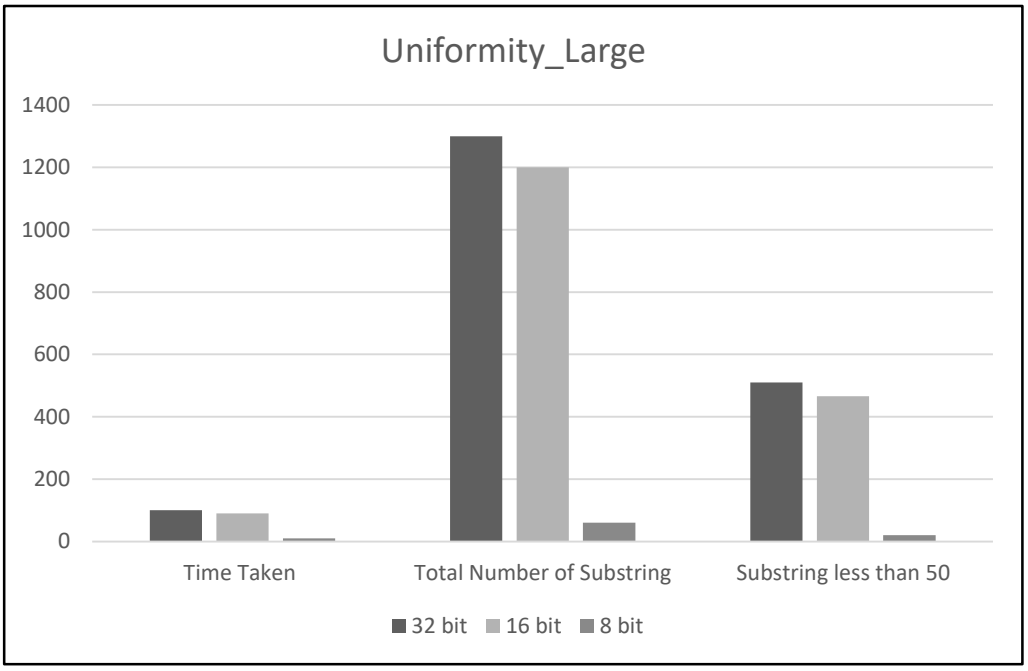


Fig 6.3(c)

Fig 6.3(a) shows the comparison of size of input we get after the second part (i.e. size of unique character and separator).

Fig 6.3(b), (c) shows the comparison of time taken, total number of substring formed and substring which are repeating less than 50 for 32 bit, 16 bit and 8 bit for two large input test file Uniformity_large and Uniformity_large_M2.

If the redundancy of data is more than using 32 bit mapping is preferable because due to redundancy substring formed by 32 bit and 16 bit will be same, so the time complexity will be approximately same but we can cover more rectangles if we use 32 bit mapping.

Repetition of data Comparison

In this repetition, data is of two types-

First, when substring identified is less than 10 characters/rectangle. Second, when substring identified is of between 10 to 20 characters/rectangles.

Comparison of data repetition between 16bit and 32bit for Uniformity_large.

Data Of Repetition			
1. When each substring has less than 10 character/rectangle			
	Each character is of 32 bit	Each character is of 16 bit	Each character is of 8 bit
Substrings which is Repeating (times)	For 32bit Mapping (times)	For 16bit mapping (times)	For 8bit Mapping (times)
In between 50 to 100	236	199	1
In between 100 to 200	103	107	4
In between 200 to 300	35	35	1
In between 300 to 400	25	23	1
In between 400 to 500	48	51	0
More than 500	197	195	34
2. When each substring is between 10 to 20 character/rectangle			
	Each character is of 32 bit	Each character is of 16 bit	Each character is of 8 bit
Substrings which is Repeating (times)	For 32bit Mapping (times)	For 16bit mapping (times)	For 8bit Mapping (times)
In between 50 to 100	73	61	0
In between 100 to 200	24	26	0
In between 200 to 300	4	4	0
In between 300 to 400	9	9	0
In between 400 to 500	2	2	0
More than 500	0	0	0

Table 6.4 Data of Repetition Comparison for Uniformity_large

Data Of Repetition 2			
1. When each substring has less than 10 character/rectangle			
	Each character is of 32 bit	Each character is of 16 bit	Each character is of 8 bit
Substrings which is Repeating (times)	For 32bit Mapping (times)	For 16bit mapping (times)	For 8bit Mapping (times)
In between 50 to 100	497	414	1
In between 100 to 200	228	219	6
In between 200 to 300	75	69	7
In between 300 to 400	122	109	3
In between 400 to 500	22	18	5
More than 500	131	119	47
2. When each substring is between 10 to 20 character/rectangle			
	Each character is of 32 bit	Each character is of 16 bit	Each character is of 8 bit
Substrings which is Repeating (times)	For 32bit Mapping (times)	For 16bit mapping (times)	For 8bit Mapping (times)
In between 50 to 100	133	82	0
In between 100 to 200	37	19	0
In between 200 to 300	6	0	0
In between 300 to 400	0	0	0
In between 400 to 500	0	0	0
More than 500	0	0	0

Table 6.4 Data of Repetition Comparison for Uniformity_large

Comparison of data repetition between 16bit and 32bit for Unifromity_large_M2

As the dimensions are magnified the probability of a substring repeating less than 100 times has increased while probability of substring repeating more than 500 times has decreased.

Increased in probability of a substring repeating less than 100 times is approx. 2 times while decreased in probability of substring repeating more than is decreased by 0.6 times only. Comparison Results for uniformity_large_M2 is in Comparison_5.

Chapter 7

Conclusion & Future Scope

Our proposed algorithm for Part 1 effectively reduces the number of iteration through the data for converting the non-monotone polygon to monotone polygon. Hence, reducing the time complexity be half.

For Part 2 we can see that 8 bit mapping is not useful as it is not able to map enough number of rectangle for mapping and while observing between 32 bit mapping and 16 bit mapping we can deduce that if the redundancy of data is more than using 32 bit mapping is preferable because due to redundancy substring formed by 32 bit and 16 bit will be same, so the time complexity will be approximately same but we can cover more rectangles if we use 32 bit mapping.

In Part 3 we concluded that the time complexity of compression algorithm depend mainly on the number of substring formed so it does not make much difference whether we are using 16 bit mapping or 32 bit mapping if the increase in substring formation is approx. same.

In the future, we may try to test our algorithm by increasing the value of bit mapping to see what is increase in time complexity and whether the number of substring formation is increasing or decreasing but if we have rectangle of maximum dimension 65535 X 65535 then it is better if we use 32 bit mapping only because increasing the mapping value will not identify any new repeating pattern so it will be of no use. Other area where we can work on is the substring match algorithm which check whether two substring formed are same or not because time complexity of third part mainly depend on the substring formed and its comparison so optimising that will reduce the time complexity.

References

[1] Sahni, San-Yuan Wu1and Sartaj. "Fast Algorithms to Partition Simple Rectilinear Polygons".

[2] https://en.wikipedia.org/wiki/Monotone_polygon.

[3] Preparata, Franco P.; Supowit, Kenneth J. *(1981),* "Testing a simple polygon for monotonicity", Information Processing Letters.

[4]K. D. Gourley and D. M. Green, "A Polygon-to-Rectangle Conversion Algorithm", IEEE Computer Graphics, Vol 3, No. 1, Jan/Feb 1983, pp. 31-36.

[5] De Berg, van Kreveld, Overmars, Schwarzkopf. Computational Geometry Algorithms and Applications. 2nd edition, Springer-Verlag. ."

[6] K. Abrahamson, on the modality of convex polygons. Discrete & Computational Geometry, vol. 5, pp. 409-419, 1990.

[7] Arkin, E., Chew, P., Huttenlocher, D., Kedem, K., Mitchell, J. An efficiently computable metric for comparing polygonal shapes. IEEE Transactions on Pattern Analysis and Machine Intelligence. 13(3):209-216. March 1991.

[8] https://en.wikipedia.org/wiki/Bit_manipulation.

[9] "Arithmetic operators - cppreference.com". en.cppreference.com.

[10] Cohoon, James, John Kairo, and Jens Lienig. "Evolutionary algorithms for the physical design of VLSI circuits." Advances in evolutionary computing. Springer Berlin Heidelberg, 2003. 683-711.

[11] https://en.wikipedia.org/wiki/Data_compression.

[12] B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In STOC, pages 71–80, 1993.

[13] https://en.wikipedia.org/wiki/LZ77_and_LZ78.

[14] B. Commentz-Walter. A string matching algorithm fast on the average. In ICALP, pages 118–132, 1979.

[15] Text Compression Algorithms - A Comparative Study, S. Senthil and L. Robert, ICTACT JOURNAL ON COMMUNICATION TECHNOLOGY, December 2011, Volume: 02, Issue: 04.

[16] D. Breslauer. Dictionary-matching on unbounded alphabets: Uniform length dictionaries. Combinatorial Pattern Matching. 184, 1995.

[17] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two-dimensional pattern matching. SICOMP: SICOMP: SIAM Journal on Computing, 23, 1994

[18] https://en.wikipedia.org/wiki/Pattern_matching

[19] "A Universal Algorithm for Sequential Data Compression". IEEE Transactions on Information Theory.

[20] Ming-Bo Lin, Jang-Feng Lee, G. E. Jan, (2006) “A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture” VLSI IEEE Transactions,Vol.14, pp925-936.