Universidad de Concepción
Dirección de Postgrado
Facultad de Ingeniería -Programa de Magíster en Ciencias de la Computación

DETECCION DE PATRONES GEOMETRICOS MEDIANTE LA EXTRACCION DE SIGNATURE (GEOMETRIC PATTERN DETECTION USING SIGNATURE EXTRACTION)

CRISTIAN ESTEBAN ANDRADES MUÑOZ
CONCEPCIÓN-CHILE
2015

Profesor Guía: Andrea Rodríguez Tastets
Dpto. de Ingeniería Informática y Ciencias de la Computación, Facultad de Ingeniería
Universidad de Concepción

DETECCIÓN DE PATRONES GEOMÉTRICOS MEDIANTE LA EXTRACCIÓN DE *SIGNATURE*

(GEOMETRIC PATTERN DETECTION USING SIGNATURE EXTRACTION)

por

Cristian Esteban Andrades Muñoz

Patrocinante: Andrea Rodríguez T

Tesis presentada
para optar al Grado de

MAGÍSTER EN CIENCIAS DE LA COMPUTACIÓN

Departamento de Ingenería Informática y Ciencias de la Computación

UNIVERSIDAD DE CONCEPCIÓN

Concepción, Chile
Octubre, 2013

Acknowledgments

En primer lugar quisiera agradecer a la comisión evaluadora de este trabajo de tesis, Nancy Hitschfeld y Leo Ferres. Igualmente, agradecer a Synopsys Inc. y al Dr. Charles Chiang por su valioso aporte y orientación como experto en el área. Particulares agradecimientos al docente guía de esta investigación, Andrea Rodríguez, por su apoyo, paciencia, y formación tanto académica como personal. Finalmente, a mi pareja y pilar, Mariana Soto, por estar ahí en cada momento.

Resumen

En la evolución constante de los procesos de manufactura de circuitos, la detección de configuraciones conflictivas se ha convertido en una tarea crucial en el proceso de diseño y producción. Estas configuraciones pueden provocar fallos en el circuito pues son propensas a sufrir distorsiones en las etapas de fabricación, ya sea por efectos de estrés de materiales, baja tolerancia a procesos de impresión como la fotolitografía, entre otros. La investigación en torno a esta problemática ha dado lugar a diferentes enfoques y técnicas que se utilizan para obtener soluciones eficaces que permitan a los diseñadores, ya sea a evitar el uso de configuraciones conflictivas, como a corregirlas o eliminarlas cuando aparecen en un diseño tras el uso de herramientas automatizadas. El problema de fondo es que estas técnicas, que suponen un coste adicional en el proceso de diseño, no siempre son eficaces, sobre todo desde el punto de vista de tiempos de ejecución.

El siguiente trabajo de tesis de magister amplía el trabajo realizado para la obtención de mi título profesional de Ingeniero Civil Informático. En dicho trabajo, se extrae cierta información relevante de regiones rectangulares de un diseño o *layout*. Esta información, llamada *signature* o firma de la región, es posteriormente utilizada para decidir si una región contiene o no una cierta configuración que se desee eliminar. Típicamente, estas configuraciones corresponden a patrones geométricos que, dados los procesos de manufactura, tienen una alta probabilidad de dar origen a una falla, conocida como *hotspot*. En otras palabras, un *hotspot* es una configuración de componentes electrónicos que provoca un mal funcionamiento del circuito. De forma adicional, este trabajo refina el trabajo anterior respecto a la estructura de datos utilizada, redefine la *signature* para patrones que tengan una dimensión distinta a la ventana de análisis, y realiza una evaluación exhaustiva de tiempo y calidad de los resultados. El objetivo principal de este método es reducir los tiempos de ejecución necesarios en las etapas de detección de *hotspots* al disminuir las porciones del diseño que deben ser analizadas usando técnicas más complejas.

Para ello se utilizan algoritmos que transforman representaciones geométricas expresadas como reglas espaciales a representaciones expresadas como grafos dirigidos, los que son finalmente utilizados para obtener representaciones vectoriales de dichas representaciones geométricas. Estas representaciones vectoriales, denominadas la *signature* de un patrón geométrico, son almacenadas en estructuras indexadas para permitir posteriores búsquedas en base a índices. El trabajo presenta evaluaciones experimentales de diversas estructuras indexadas basadas en arboles binarios de búsqueda, tablas hash, y combinaciones de ambos tipos. Los experimentos muestran tiempos de ejecución que permiten disminuir los tiempos de ejecución que actualmente se requieren para la detección de patrones en diseños de circuitos integrados, así como uso de memoria que permiten la ejecución del proceso en computadores personales. Las estructuras que utilizan una combinación de estructuras indexadas mostraron ser la que ofrecen la mejor relación entre tiempos de ejecución y memoria requerida.

Abstract

Due to the constant improvement of the circuit manufacturing technology, the detection of conflictive configurations has become relevant, being an important part of the design and production process. These conflictive configurations can lead to circuit failures, which can be produced due to a series of factors that can generate distortions at manufacturing stages. These factors can be, for example, the stress of materials, low tolerance to printing processes as photolithography, among others. The investigation around this problem has given rise to different approaches and techniques used to obtain effective solutions, which allow the designers to avoid the use of conflictive configurations, or correct or remove them when they appear in a design after the use of automated tools. The background problem is that these techniques, which suppose an additional cost to the design process, are not always efficient, mostly from time point of view.

This Master's thesis improves and extends the work done to obtain the professional title of *Ingeniero Civil Informático*. It extracts some selected information, referred as *signatures*, from rectangular regions of a circuit design or *layout*. These signatures are then used to decide if a region contains or not a configuration that want to be removed. Typically, these configurations correspond to geometric patterns that, due to the manufacturing process used to build an IC, have a high probability to become a failure, which is known as *hotspots*. In other words, a *hotspot* is an electronic component configuration that causes a malfunction in the circuit. Additionally, this work refines the previous work regarding the data structure used, redefines the *signature* for patterns having different size with respect to the window of extraction, and makes a comprehensive evaluation of time and quality of results. The principal objective of this method is to reduce the time cost of the *hotspot*-detection process by filtering out portions of the layout that need to be analyzed using more complex techniques.

To achieve this objective, some algorithms are used. These algorithms transform a spatial-rule representation of a geometric pattern into an oriented-graph representation, which are finally used to obtain a vectorial representation of the geometric pattern. This vectorial representation, called the *signature* of the geometric pattern, are stored in a indexed structure that allows later lookups based on indexes. This thesis work presents an empiric evaluation of various different indexed structures based on binary-search trees, hash tables, and combinations of both types. The experiments show execution times that allow to reduce the execution times currently needed to detect geometric patterns in integrated circuit designs, as well as the memory required allows to operate the process in personal computers. The data structures that use a combination of indexed structures have shown to be the ones that offer the better balance between execution times and memory used.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction.

1.1 Background.

A typical integrated circuit or IC is built in a chip, creating layers of different materials such as metal or polysilicon[1]. Geometrical patterns are then printed on these layers using different printing techniques, giving rise to electrical components whose interactions will define the circuit behavior. Designing a circuit requires to have knowledge about the interaction between layers for a certain manufacturing technology, as well as the limitations that these manufacturing processes impose.

The principal objective of a designer is to put as many electrical components in the less area as possible, achieving a correct circuit behavior. A good example of what is an IC is a microprocessor for which it is very important the amount of information that can be processed and the processing speed of the microprocessor. Because the enormous quantity of electrical and wire components needed to build an IC, the schematization problem, known as *layout problem*, becomes relevant.

Current technology for nanoscale IC construction has certain problems inherent in the manufacturing process that require special attention from designers. As going through the circuit design process that involves determining which pattern will be included in a design, in which layer and position they will be included, and do so trying to make a design as compact as possible, designers have to prevent the appearance of *hotspots*. The prevention can be achieved by detecting the occurrence of conflictive patterns in a design, and correcting or removing them before the design goes to production. This detection becomes critical below the 90 nanometers technology [13] as a reason of printing problems that cause variation between a design and its physical circuit, such as focus fixation or light exposure in the photolithography process. These variations can lead to *bridging* between electrical components, and *necking* or *line-end*

[1]Also called Polycrystalline silicon.

shortening in its dimensions when they are printed (see figure 1.1). These anomalies cause the circuit to fail in contrast to some of the specifications under which the circuit was designed, reason why they are known as *yield detractors* or *hotspots*.

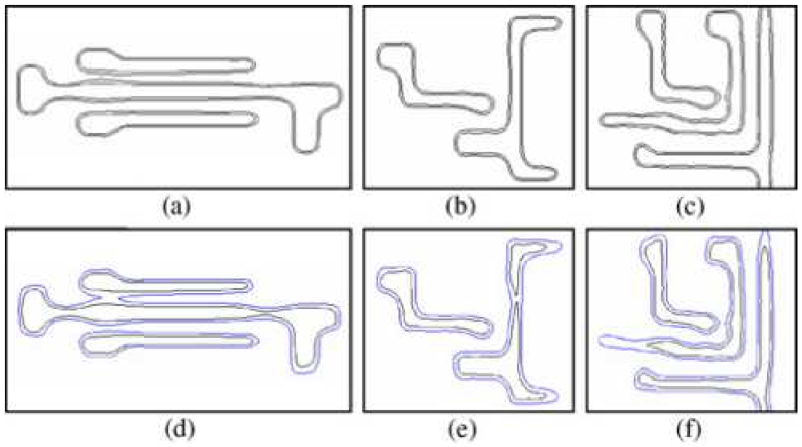


Figure 1.1: Examples of manufacturing variations. [19]

(a)-(d) Bridging, (b)-(e) Necking, (c)-(f) Line-end shortening.

Design-rule checking (DRC) was introduced as a first attempt to help designers to detect and avoid *hotspot* occurrences in designing stages. DRC is a major step during physical verification signoff of the design, where the designer defines a set of geometric restrictions and connectivity rules, known as *Design Rules*, that the circuit must satisfy (see figure 1.2). These rules seek to ensure sufficient margins according to the variability in the printing processes, reason why they are specific to a certain manufacturing process and may not be generalizable.

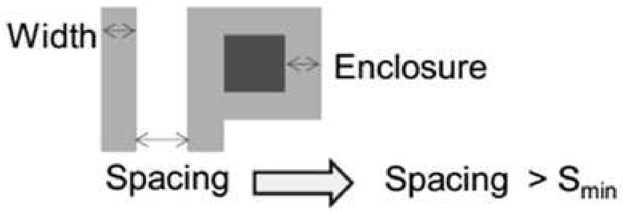


Figure 1.2: Basic DRC checks. [4]

Width, spacing, and enclosure.

What is common is that, with the advance of manufacturing technology, the number of necessary rules, and its combinatorial explosion due to the significant increase of elements in a circuit, have increased the complexity of this technique. A standard practice is to reduce the rule number relaxing the set. This means to pass over some complex rules, which reduces the complexity of the verification process as expenses of yield. This decision may cause that some potential *hotspots* be overlooked, especially when the conflicting interactions are not local. For this reason, new complementary techniques were introduced. Some of them take into account local interactions, while others are capable of taking into account more global interactions.

These techniques suppose a high additional cost in the verification process, especially due to the dimensions of current designs and the density of components that the new technologies allow, increasing the apparition of potential *hotspots* and, therefore, the patterns that must be detected.

1.2 Motivation.

Semiconductor fabrication plants, commonly called *fabs*, usually use design rules to represent process-*hotspots* and a typical design rule checker to detect such *hotspots*. However, this representation has been found to be inadequate [6], so the last few years there has been extensive work in the area of fast process simulation to aid DRC during the physical design stages. For example, industry has been using lithography *hotspot*-detection processes based on aerial image simulators [22, 16]. The problem is to find new techniques that complement verification techniques that are currently used by *fabs*, at a low computational cost, usually measure in terms of runtime.

A previous work in [3] defined signatures to pre-filter regions of a layout from a *hotspot*-detection process. These signatures consist in some summarized information of a portion of the layout. Once the *signature* is obtained and stored, it is then used to decide if the portion of the layout contains or not a certain pattern to be identified. This identification process is used to modify or remove these patterns to avoid the occurrence of yield detractors or *hotspots*. The experiments shown in [1] laid the theoretical basis on which it is asserted that the use of signatures is a viable technique that allows this decision with a low rate of false positive. However, the proposed method still had various limitations and improvements to be done. Of particular interest are a more detailed *signature* extraction process and a study of different storage structures. Additionally, a rather simplistic limitation was the fact that to obtain good results in terms of accuracy (i.e., low rate of false positive), the pattern to be searched should match the size of the portions or windows upon the signatures were extracted. This limitation means that the method was practically unusable in production processes because it is really very difficult and unrealistic to know the dimensions of a potential *hotspot* beforehand. Even more, the dimensions of same patterns could vary as printing technologies do.

1.3 Hypothesis.

The hypothesis of this work is that it is possible to construct a *signature* that characterizes a certain configuration of geometries (from now the pattern), and use that *signature* to decide if this configuration is or is not contained in a certain area of a circuit design. To use this *signature* in a realistic situation, it must allow the decision even if the size of the pattern and the area of design are not exactly the same. It also must allow the detection of patterns even if there are rotations between the pattern and the design region.

1.4 Objectives.

The objectives of this thesis are:

1. Define a *signature* over configurations composed of rectangular geometries. These configurations can be patterns or portions of layouts. The computation of these *signature* should consider time, and as a second priority, space costs.

2. Design an indexing structure that stores *signatures* and the identification of portions of a layout that match the signatures.

3. Design a search method over the indexing structure that use the *signature* of a pattern as a search key.

4. Perform an experimental evaluation of the quality of the *signature*, the time and space cost of the indexing structure and the time cost of the search process.

1.5 Methodology.

The methodology followed by this work can be summarized in three basic steps that aim to answer the following questions: How the *hotspot* detection is addressed today? How can we contribute to improve the efficiency on the *hotspot*-detection process?, and How can we evaluate our contribution?

The first step is to present a review of the problem, determining the principal approaches in the state-of-the-art and their limitations. From this review, we can determine some useful elements to be used in our work, delimiting the contribution. From this first step, we chose approaches that result to be more similar to our previous done work in [1]. The review include works found in abstracts, papers and patents published in academic and commercial journals, bibliographic databases, and Internet search engines.

The next step is to propose a *signature*-based preprocessing and searching to filter out portions of a layout from *hotspot*-detection process. This includes to propose a data model to represent layouts and patterns and to design an indexing structure.

The last step is to determine the data analysis to be performed and what the evaluation criteria are. This step needs to set up the experimental evaluation to measure the evaluation criteria. Because a standard data analysis method has not been established nor decided yet from companies engaged in the area of this investigation, we use typical measurement form information retrieval and execution time criteria. These evaluations were performed by using our own implementations because more sophisticated analysis tools are typically distributed under paid licenses and need to be calibrated to a certain production process.

1.6 Organization.

The organization of this document is as follows. Chapter 2 provides a brief review of related works, presenting techniques for *hotspots* detection that are used today, and resuming the contribution of this master thesis. Chapter 3 discusses the experimental evaluation, its results and conclusions, while the final chapter closes the work with general conclusions and future research directions.

Chapter 2

Related works.

In the literature we can distinguish four approaches for *hotspots* detection, each of them with its advantages and disadvantages. These four approaches are the *hotspot* detection using lithography simulation, the *hotspot* detection using pattern recognition, the *hotspot* detection using learning machine, and the *hotspot* detection using graph-oriented techniques. In this chapter we present a brief review of these approaches.

2.1 Lithography simulation.

The real process of printing an IC involves more than only the design and the wafer, which is the surface where the IC is printed. As was said before, an IC is built layer by layer which are superposed obtaining a third-dimensional product from bi-dimensional representations. These representations include geometric shapes and are known as the design. Typically, the designs are constructed using CAD[1] tools and are used to build what is known as *mask.* A mask is used in the photolithographic process to print the design on the wafer.

As we can see in [12], to build a layer it is necessary to deposit material over a silicon layer and then remove part of it. The deposition process typically used a method called *Chemical Vapor Deposition* (CVD), while the removing process typically uses etching substances, leaving exposed some areas of a layer. To do this etching process, it is necessary to ensure that the areas that want to be maintained will not be etched by the acid. This is achieved by reinforcing the materials using light, from which this process receives the name of photolithography. A material known as photoresist is used and some regions are exposed to light. The areas with incidence to light can become resistant to the acid or soluble to it. In this way, geometrical shapes can

[1]CAD stands for Computer Aided Drafting

be printed on a layer. The problem is that current dimensions of components are smaller than the frequency of the light used in this process. Because of this reason, the light should pass through a lens and mask that filter out the light exposition in some regions of the layer.

The resolution of a process of optical lithography, i.e., the ability to project a clear image of a small feature on the wafer, is limited by the wavelength of light and the numerical aperture (NA) of the reducing lens used for light exposure [7, 8]. As the minimum widths in Ultra Large Scale Integrated (ULSI) circuits reach resolutions smaller than 90nm, the difficulty of achieving high fidelity printing circuits increases, even when the NA significantly increases and the wavelength decreases [5].

The minimum width of the features that a projection system can print is given approximately by the formula 2.1, where CD corresponds to the minimal width that can be achieved while printing onto the wafer, k_1 corresponds to a coefficient that encapsulates different factors related to the photo-lithographic process (usually varies between $0,3$ and $0,4$), NA corresponds to the NA of the lens view from the wafer, and λ is the wavelength of the light.

$$CD = k_1 \cdot \frac{\lambda}{NA} \tag{2.1}$$

Formula 2.1: *Critical dimension* of the features printed by a projection system.

Currently, laser light with wavelength between 248 nm and 193 nm are used. Photolithographic processes for low minimum widths require narrow margins for focus and exposure, resulting in high optical proximity effects as deviations in the critical dimensions of the prints [9]. These difficulties have led to the semiconductor industry to use complex conditional rules and recommended rules for specifying geometric constraints in the design stages [11], Resolution Enhancement Technology (RET) techniques, Optical Proximity Correction (OPC) [9].

OPC is a photolithography enhancement technique commonly used to compensate image errors due to diffraction or process effects. It is mainly used in the semiconductor industry due to the limitations of light techniques to maintain a correct edge placement, relative to the original design, into the etched image on the silicon wafer.

These projected images appear with irregularities such as line widths that are narrower or wider with respect to the design. The irregularities can be compensated by changing the geometries on the mask used for imaging, and here is when OPC is used. OPC corrects these errors by moving edges or adding extra polygons to the pattern written on the mask or *photomask*. This may be driven by pre-computed look-up tables based on width and spacing between features, known as *rule based OPC*, or by using compact models to dynamically simulate the final pattern and thereby drive the movement of edges, typically broken into sections to find the best solution, known as *model based OPC*. The objective is to reproduce, as well as possible, the original layout drawn by the designer in the silicon wafer. This is achieved through iterative design modifications that are usually based on rules or models of the manufacturing process used by the *fab*.

Some process models have incorporated analysis and correction tools such as complete circuit simulators [14]. Although lithographic simulations generate accurate images of a design allowing robust verifications, these approaches have some limitations that make them hard to use in the practice [6].

1. Lack of information about subsequent processes, making it impossible to accurately model and calibrate some stages as OPC corrections. Moreover, simulations based on lithographic images usually detect regions that could be easily corrected by using masks-synthesis techniques, causing an over-estimation that slows the process of routing.

2. High computational cost of some models that are difficult to incorporate during physical designing stages, e.g., calculation of stress in metal components.

One approach that prevails today for *hotspots* detection is to predict them using a rigorous simulation post-OPC[2] along with customer-defined checks (see figure 2.1). Thereby, the configurations that don't meet the checks are marked or classified as *hotspots* and are corrected in accordance to this classification. Unfortunately, a complete simulation of an entire circuit design is computationally expensive, increasing greatly the time a circuit passes from the design stages to the production stages.

[2]OPC stands for Optical Proximity Correction.

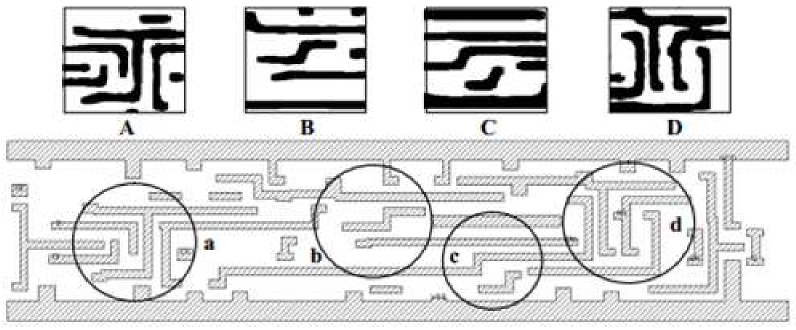
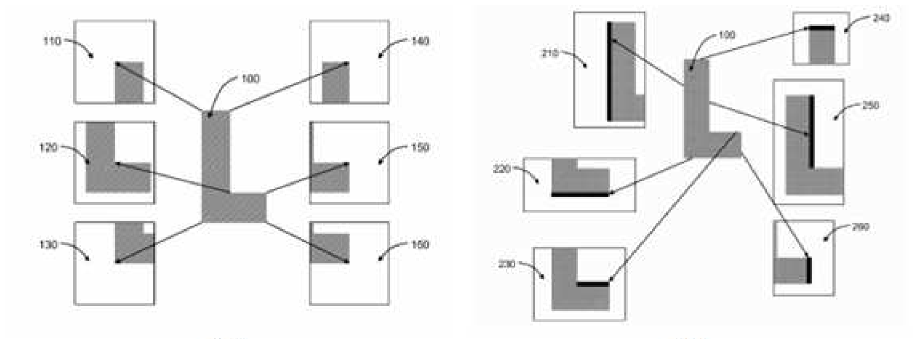


Figure 2.1: Example of a lithographic simulation post-OPC for an 45 nm design. [20]

a, b, c and d are design views; A, B, C and D are lithographic simulations.

To overcome the cost of a full simulation, there are different techniques that aim to break up the complexity of this process. The work presented in [15] extracts *situations* of a design. A situation is a collection of shapes within a radius of extraction, which are described based on corners and edges. This generates a collection of figures among which may exist overlaps. These *situations* occur depending on some characteristics of the geometry to be analyzed, and can be used to improve OPC corrections. Figure 2.2 shows an example of the extraction of situations based on borders and a predefined radius. The radius could be expressed as Manhattan distance, Euclidean distance, among others, and typically corresponds to the radius of optical influence of the manufacturing tools. In the figure 2.2(a), the situations 110-116 are extracted from the polygon 100, based on corners and a r-Manhattan radius of extraction, which results in $2r \times 2r$ square situations. Similarly, the figure 2.2(b) shows the situations based on borders for the same polygon and the same radius. This result in situations of $2r+x \times 2r+y$, where x is the length and y the height of the border around which the situation has been extracted. Interesting is to see that situations are described within a radius, and, therefore, are part of a larger geometry with its own characteristics. These situations are extracted in a hierarchical order, so that the design to be analyzed can be conceived as a partition of cells. The process traverses through this hierarchy of cells extracting a collection of geometries, where each cell instances can be seen as a cell instance of the cell in the top level.



(a) (b)

Figure 2.2: Extraction of *situations*. [15]

(a) corner based y (b) edge based.

The drawback of this approach is to define the situations and run simulations on each of them to make the necessary changes and then merge these situations to get the entire design again, but with the changes done. To avoid performing unnecessary simulations, the overlapping areas must be handled properly, and yet the separations of a geometry can generate loss of information of the interactions between components that are beyond the location of the partition.

2.2 Detection using pattern recognition.

In the literature, pattern recognition for *hotspot* detection can be addressed in two ways. One of them is to represent patterns as Design Rules (DRC). The other way is to represent patterns as spatial configurations, typically as images.

Although DRC was not introduced for *hotspot* detection but to prevent the appearance of them, there are approaches that leverage the DRC characteristics to improve the accuracy in the *hotspots*-detection processes. 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 yield, DRC has evolved from simple dimension measurement and boolean checks, to more complex rules that modify existing features, insert new features, and check the entire design for process limitations such as layer density. However, DRC is a computationally very expensive task. Usually DRC checkers will be run on each sub-section of the ASIC[3] to minimize the number of errors that are detected at the top level. If DRC runs on a single CPU, customers may have to wait up to a week to get the get the result for modern designs. Most design companies require DRC to run in less than a day to achieve reasonable cycle times because the DRC be likely run several times prior to the design completion. As complex conditional and recommended rules are added to compensate some deficiencies, the number of needed rules to represent a layout can result in an explosion of the rule libraries. Hence, depending entirely on DRC to detect potential *hotspots* usually slows down the design process. Even more, the systematic use of recommended rules can bring the density of a 90 nm design to the equivalent of a 130 nm design, wasting most of the benefit that a 90 nm design supposes. For this reason, the set of rules is usually relaxed and some *hotspots* can appear. However, DRC can still be useful to detect *hotspots* converting some topological characteristics of conflictive patterns into rules and then analyzing and comparing DRC logs to identify potential *hotspots* [10, 15].

The work presented in [4] extracts topological characteristics of a certain pattern and converts them into design rules. This extraction considers different orientations of the pattern that include the four rotations $(0°, 90°, 180°, 270°)$ and the mirroring over X and Y axis, respectively. Once these critical topological features have been extracted and converted into design rules, a DRC process is applied to find regions of the design that fit the rules extracted. This occurs in two filtering stages, where the first stage indicates potential regions, while the second stage verifies the exact locations. To extract the topological relations of a pattern, they extend the representation given in [2] to obtain a modified transitive closure graph or MTCG. The original representation or TCG is used to represent compact configurations, using a pair of restriction graphs C_h and C_v (horizontal and vertical restrictions, respectively), to establish the geometric relationships. However, as we can see in figure 2.3, design patterns cannot be in a compact form because the spacing between geometries. This spacing should be considered as it is essential for the detection of *hotspots* because it also represents topological characteristics of the pattern. To properly consider these spaces in the TCG representation, the pattern is partitioned into boxes. As seen in

[3]ASIC stands for Application Specific Integrated Circuit

figure 2.3, in the horizontal partition (Tiled pattern (H)), the horizontal edges of the geometries contained in the pattern are projected until they intersect a vertical edge of a geometry and such partition will correspond to a box (boxes a-d and A-E). The same way occurs for the vertical partitioning of the pattern, reversing the orientation of the projected geometries. So, a compact representation of the pattern is obtained, where there exist block boxes provided by the pattern geometries (a-c boxes in figure 2.3(H)) and spacing boxes (A-E boxes in figure 2.3(H)).

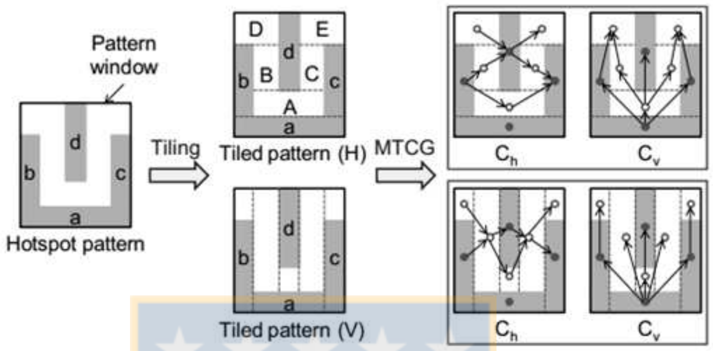


Figure 2.3: Construction of a MTCG graph. [4]

After the pattern has been partitioned into tiled patterns, the MTCG graph is built. Here, the vertexes represent block boxes (black dots in figure 2.3) or space boxes (white dots in figure 2.3), and the directed edges represent topological relationships between these boxes. In a horizontal contraint graph, a directed edge is added between adjacent boxes if the projection on the X-axis produces a superposition, starting from left to right. This is similar for vertical restrictions graph and the Y-axis, from the bottom to the top of the pattern.

In the first topological features extraction, lengths and widths of blocks are extracted as well as distances between two adjacent boxes and diagonal relations between two convex corners. These features are then traduced to rules, e.g., width and spacing rules shown in figure 1.2. Being written as design rules, these characteristics are expressed as inequalities.

Once the topological features of a pattern are extracted and are written as design rules, and because the representation of a pattern in MTCG is unique, the method proceeds to the verification stage. Here the extracted design rules are compared to the DRC rules under which the design was built. The idea is to determine whether any of these rules are violated. If this occurs, the location is verified in more details to see if there exists a *hotspot*. Something similar occurs in [10], where all the lengths of edges and the distances between geometries are extracted as topological features. At this analysis stage, a searching graph is built. This graph stores all the locations reported by a DRC comparison. Subsequently, the searching graph is traversed to identify potential *hotspots*.

When patterns are represented as spatial relationships for *hotspots* detection, some relevant information of conflictive patterns are usually stored in libraries. These libraries are queried to find *hotspots* occurrences scanning the whole design using pattern matching techniques. This means that these techniques have all the benefits of current pattern matching algorithms. However, even using algorithms for fuzzy geometries [6], they suffer of poor accuracy when previously unknown patterns have to be detected. Moreover, the pattern libraries can be updated to include more information about conflictive patterns as the system progresses, but as the library grows, the run-time advantages of the method decreases.

In [6], a worm-like movement is used to analyze all possible windows within a design. Each window is converted into a matrix of binary values, as shown in figure 2.4, where the value of each cell is 1 if the cell overlaps a geometry, and 0 otherwise. This process occurs in two stages, the first stage with a low resolution grid, to filter potential regions corresponding to conflicting patterns, and in a second stage with a higher grid resolution, for a more accurate analysis of selected regions at the first stage.

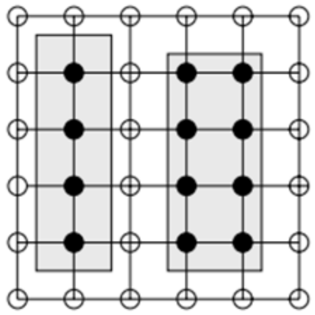
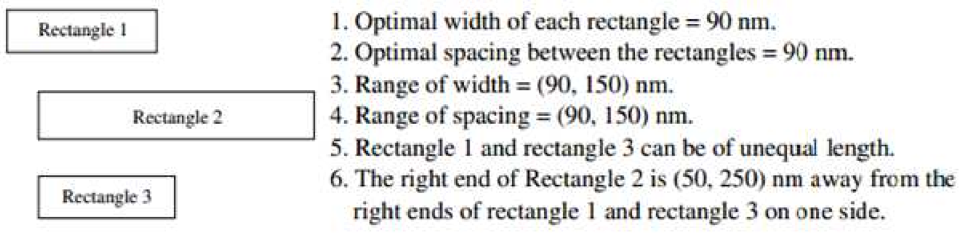


Figure 2.4: Binary matrix representation. [6]

For the patterns to be detected, a similar procedure is followed, with some differences. To properly handle the sizes of the libraries, a new representation called *range patterns* is introduced (see figure 2.5).



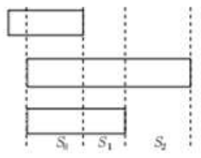
1. Optimal width of each rectangle = 90 nm.
2. Optimal spacing between the rectangles = 90 nm.
3. Range of width = (90, 150) nm.
4. Range of spacing = (90, 150) nm.
5. Rectangle 1 and rectangle 3 can be of unequal length.
6. The right end of Rectangle 2 is (50, 250) nm away from the right ends of rectangle 1 and rectangle 3 on one side.

Figure 2.5: *Range pattern* Rocket. [6]

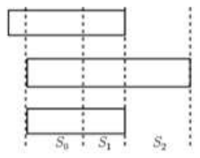
Unlike representing every possible conflicting pattern, establishing the absolute dimensions of the geometries contained in it, a *range pattern* representation allows grouping a set of patterns with similar geometries into a single pattern. This grouping occurs through the definition of ranges in the dimension of their geometries. For example, the *range pattern* in the figure 2.5 represents both the pattern for which the length of the rectangle 1 is 90nm, and the pattern for which the length of the rectangle is 91nm, and so on until 150nm. Without this compact representation, each of these instances of a *range pattern* should be specified independently. Explicitly listing each instance, which will be referred from now on as absolute patterns, leads us to obtain 60 different patterns from varying just one of their geometries. Thus, the representation by compacting *range patterns* allows the decrease of memory requirements for libraries in *hotspots*-detection processes. Additionally, due the characteristics of the process used for handling *range patterns*, it is possible to decrease the number of runs required to detect two absolute patterns if they belong to the same *range pattern*.

To perform the comparison of patterns, a *range pattern* should be represented in a way similar to the design representation, i.e., as a matrix representation. To achieve this, a *range pattern* is represented in what the authors call *cutting-slice* representation. For example, as shown in figure 2.6(d), to obtain the horizontal cutting-slice representation of the pattern rocket, all the horizontal edges of the geometries are projected obtaining a set of 2D matrixes. What characterizes these 2D matrixes is that, in a 2D matrix, all rows (or columns in the vertical case) are equal. The set of

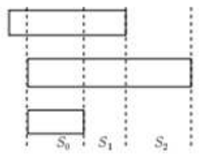
2D matrixes is called cutting-slice and in a cutting-slice two adjacent slices are not equal. In figure 2.6(d) we can see that $S_0, ..., S_4$ are horizontal slices and conform the horizontal cutting-slice of the pattern rocket. The same occurs for the vertical cutting-slice.



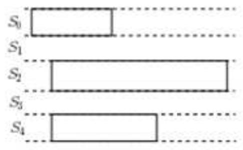
(a) R1 to the left.



(b) R1 aligns with R3.



(c) R1 to the right.



(d) Horizontal slicing direction.

Figure 2.6: Cutting-slices of the rocket *range pattern.* [6]

Once the window is represented as a grid and the pattern as cutting-slices, both are encoded as strings to perform the first detection stage. This step compares the encoding to detect when a set of geometries within the window is similar to the geometries contained in the pattern, even when the dimensions of these geometries are not exactly the same. When a possible match is detected, the window is analyzed in a second stage where the resolution of the grid is higher, and the process takes into consideration additional information related to the dimensions of the geometries for a more accurate detection.

2.3 Detection using machine learning techniques.

The techniques that use machine learning for *hotspots* detection, typically construct a classification model for patterns. Such model catalogs a pattern as *hotspots* based on the extraction of some features, and can use the classification model to predict whether future unknown patterns will give rise to *hotspots* in a certain manufacturing process. The work presented in [20] uses a machine-learning kernel (MLK) based on the extraction of critical features for *hotspot* detection. This extraction occurs in binary images that are representations of a layout. The authors propose three critical features to generate a compact representation of a certain geometrical configuration: the number of bounded rectangles (BR), T-shapes and L-shapes. This compact representation is not affected by two-dimensional transformations such as shifting, rotation or mirroring.

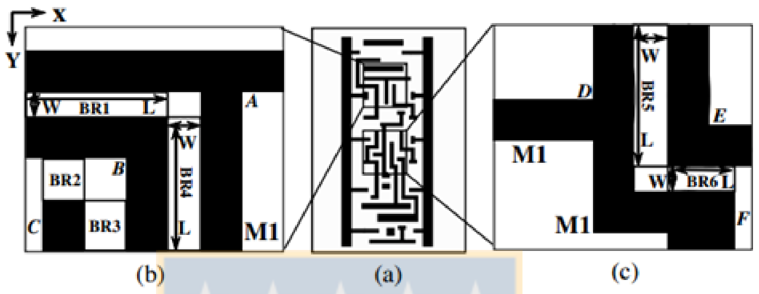


Figure 2.7: Critical features. [20]

(a) a 45nm design, (b)(c) two examples of critical features extraction.

The bounded rectangles represent the relative geometrical positioning between adjacent components using an interval representation. For example, in figure 2.7(b) we can see that BR1, BR2, BR3 and BR4 represent de relative positioning between the components (black rectangles). Each BR is represented as a vector which codifies the parameters $(W, L, (X, Y), D)$, where W and L are the width and the length of the bounded rectangle respectively, (X, Y) are the top-left coordinates of the rectangle, and D corresponds to the orientation of the rectangle. D is 0 when the width of the rectangle is measured along the X-axis, and 1 otherwise. On the other hand, T-shapes and L-shapes correspond to the number of shapes with a T-form or an L-form, respectively, that extend along both sides of the bounded rectangles. For example, in 2.7(b), the zone denoted as A is a T-shape for BR1 and BR4, the zone denoted as B is a L-shape for BR1, BR2, BR3 and BR4, while the zone denoted as C does not correspond to a T-shape nor a L-shape for BR2 and BR3.

In the extraction step, the binary image is partitioned in a set of binary images.

For each partition, a vector is obtained for each component within the partition. An ordered collection of vectors establish the metrics for the whole partition, which are then used as input for the MLK, iteratively creating a kernel to predict the presence of *hotspots* in future designs (note that the training of the MLK must be supervised). The MLK is based on an artificial neural network (ANN) whose objective is to minimize the squared error between the network prediction and the supervised decision. The work is used by the CALIBRE [4] design tool.

In [19], the authors propose a technique for *hotspot* detection that provides a full layout, feature-centric analysis. This is achieved by determining a radius of analysis, under which geometrical features are extracted (see figure 2.8). The work, which extends the work presented in [18], uses the metric features extracted from the layout not to decide whether a certain pattern is defective or not, but leaves the decision to a recursively trained and validated kernel using techniques of machine learning. This kernel, that could be based on an ANN or in a support vector machine (SVM), extends the previous work to a hierarchical learning process. The extraction process defines a radius of analysis, under which the geometries are fragmented to analyze each fragment and its neighbor fragments, defining externally and internally facing polygons.

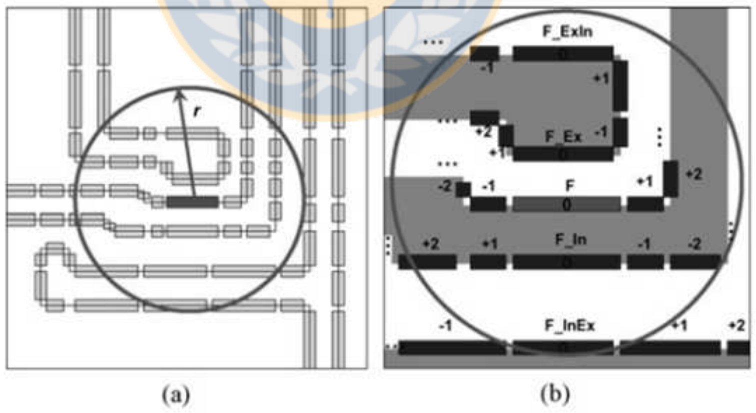


Figure 2.8: Feature extraction. [19]

(a): Efective radious of analysis. (b): Fragmented geometries with externally and internally facing polygons.

[4]CALIBRE is a lithographic simulation tool of Mentor Graphics. -http://www.mentor.com.

The characterization of geometries is built based on corners information (concave or convex), the distance between externally facing polygons, and the distance between internally facing polygons (see figure 2.9). This information is used to build a uni-dimensional vector that characterizes a fragment F. These vectors are then combined using vector operators to define a final vector V_f that characterizes the whole region of analysis. As well as the previous work, this vector V_f is invariant to rotation and mirroring, and is used as input for the SVM or ANN kernel.

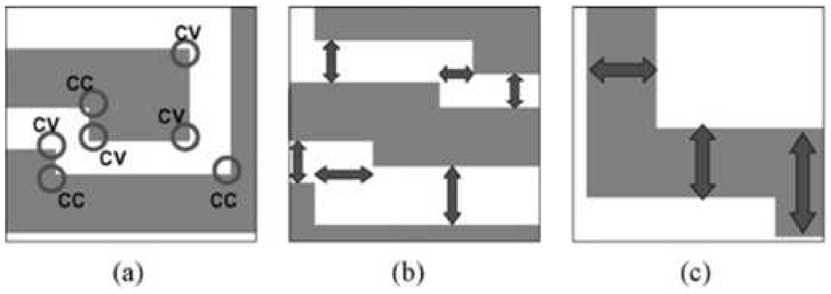


Figure 2.9: Geometrical metrics. [19]

(a): Corner bases information. (b): Distance between externally facing polygons. (c): Distance between internally facing polygons.

The problem with the previous mentioned technique is that it take into account very local relations between components. Most of similar techniques use a region of analysis, so this problem could be overcome using larger regions, but this decision could increase the runtime of the process. The work presented in [23] attempts to overcome this limitation by taking into account the information of larger regions of a design, improving the execution time by using a two-level classification. The main idea of this classification is:

1. The first level uses a supervised trained classifier to separate parts of the design with potential *hotspots*. This separation is performed using information around a centric target location, e.g., geometries of the design in its vicinity. Since these near geometries have a strong influence on candidate geometries, then geometries very similar to a pattern, predefined as potential *hotspots*, will be marked at this level and will pass to the second level of the classification for a more detailed examination.

2. The second level uses as well a previously trained classifier to examine the peripheral information of the region marked as potential *hotspot* at the first level. Thus, geometries that are not in close proximity, but still have non-negligible effects on the internal geometries will be classified as *hotspots* as well. This decision occurs because as they influence the current region, marked as *hotspots*, it is desirable to remove them to not allow influencing other regions of the design.

The classifiers are built using SVM, where the input vectors encode information from regions that are represented by the density of geometries that they contain. The figure 2.10 shows an example where a clip window centered at a candidate location is pixelated, and pixel densities are computed (note that the process needs at least two additional input parameters, i.e., region size and pixel size). An ordered list of density values forms the output vector. As we can see, this feature encoding step should not be confused with the actual pattern features such as convex/concave corners or line-ends. The goal of this representation is not to identify those geometrical features that may degrade the printability of a pattern. Instead, it aims to provide a compact representation of layout patterns, enabling an efficient measurement of pattern similarities for classification purposes. The SVM is the responsible to make the decision after the supervised training.

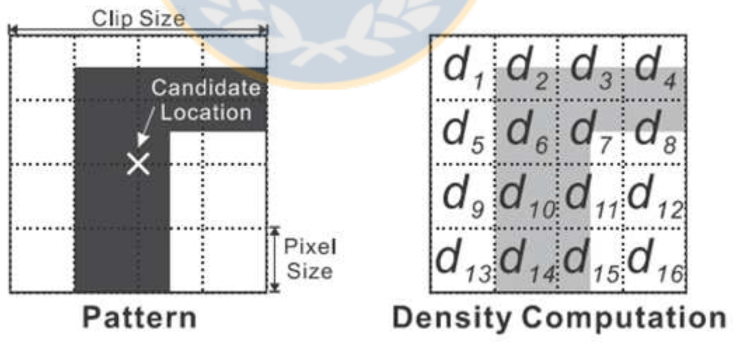


Figure 2.10: Representation based on geometries density. [23]

The vector is $[d_1, ..., d_{16}]$.

As we can see, most of the techniques that use machine learning for detecting *hotspots* require to know the *hotspots* patterns beforehand to use them in the training steps. While current methods demonstrate high accuracy, unknown patterns are classified as *hotspots* with a certain probability of certainty. The problem with these strategies is that for many manufacturing factories, to not properly detect the *hotspots* is not acceptable. This means long training and calibration steps, which can lead to high occurrences of false-positive errors.

2.4 Graph-oriented detection.

As in [4], where graphs are used to represent conflictive patterns and then these graphs are used to obtain design rules that can be used for *hotspot* detection, there are other works that use graphs with similar objectives. An example is the work presented by Kahng *et al.* in [5], where a *hotspot* filtering method derives a dual graph from an image of the pattern. The work involves the use of a graph of a certain region of the design, which reflects the variation of the critical dimensions of the geometries contained in the region. The detection is not based on the existence of patterns established as *hotspots*, but in the idea that any configuration of geometries whose interactions generate a high variation in the critical dimension (CD) could lead to the occurrence of a *hotspot*. This is similar to what occurs in the detection based on design rules, except that in this work no conditions are explicitly specified by designers. The analyzed regions are separated according to their complexity, because the greater is the complexity of the geometries that they contain, the greater is the possibility of the existence of a *hotspot*. Additionally, different types of variations in critical dimensions are separated into two types:

1. CD variations induced by corners, where two orthogonal geometries connected as a corner may vary as corner-rounding (see figure 2.11(a)).

2. CD variations induced by the proximity between geometries, where two very close geometries can suffer necking or bridging (see figure 2.11(b)-(c)).

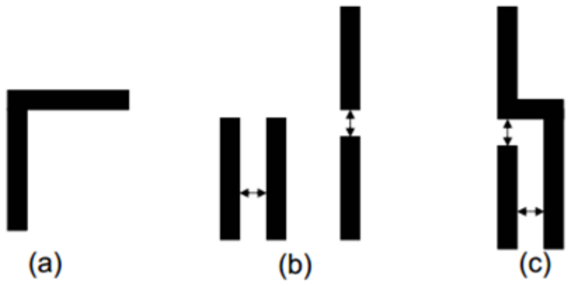


Figure 2.11: CD variations induced by the interaction between geometries. [5]

The first stage of the process is to build a graph G for regions of the design that reflects the CD variations induced by the geometries contained in the region. Given a design region L, the design graph $G = (V, E_c \cup E_p)$ consists of nodes V, corner edges E_c, and proximity edges E_p.

1. For each horizontal and vertical geometry, a node $v \in V$ is created.
2. For two orthogonal geometries connected, the corresponding nodes are connected by an edge $e \in E_c$, where the weight of the edge is a constant.
3. A proximity edge $e \in E_p$ is created between two nodes if the corresponding geometries are adjacent and have the same orientation. The weight of the edge reflects the separation between the geometries and the overlap of the width or length projections.

Figure 2.12(a) shows an example of a region which contains 7 nodes representing 7 geometries, 4 corners (dotted lines) and 5 proximities (solid lines).

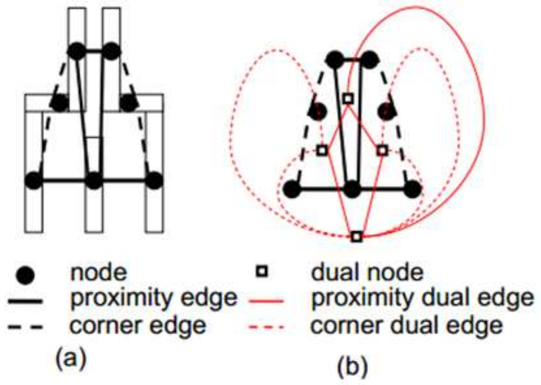


Figure 2.12: Graph and dual graph representation. [5]

The second stage consists in the planarization of the graph and obtaining its dual graph. Thus, the graph $G = (V, E_c \cup E_p)$ is converted to its dual $G^D = (V^D, E_c^D \cup E_p^D)$. The dual graph G^D of a graph G is constructed by representing each face f of G with a dual node n, which is the sum of the weights of the edges forming f. An edge e that belongs to two faces f_1 and $f_2 \in G$ is represented with a dual edge $e^d = \{n_1, n_2\}$ in G^D with the same weight of e. As the dual graph G^D exists if G is planar, that is, there are no edges that intersect, for each edge that does not fulfill the condition, the less weighted edge is eliminated. Finally the detection of *hotspots* occurs in three levels, by using a lookup table with weight values that are considered for potential CD variations. The detection at the dual graph edges level detects the *hotspots* caused by two close geometries or two geometries connected as corners. Face-level detection finds *hotspots* produced by several close geometries. Finally, the merged-face level detection finds *hotspots* produced by complex relationships between geometries. To do so, once the dual graph G^D is built, its nodes are sorted by their weights. Then, the sorted nodes that share the same geometries are merged, which means to merge the faces in G. This detection level is based on the idea that a *hotspot* is a combination of various local "bad" geometric configurations. Assuming that the effect on the CD variation is accumulative, this effect may by reflected by the weight of a dual node, that is, the total weight that conforms the face in G. However, a *hotspot* can be produced by more complex relationships between geometries that belong to different faces. For this reason, the authors consider merging nodes of the dual graph and running a detection on this merging.

Chapter 3

Signature-based indexing and searching.

Unlike previous work in *process-hotspot* detection that uses complex features in the process of *hotspot* detection, we propose to pre-process layouts in order to extract signatures that characterize windows within the layouts. The idea is to use these signatures to filter out portions of the layout that could be excluded from the *hotspot*-detection process, reducing the costs and efforts needed to detect the occurrence of *hotspots*. The *signature* can be used as search criteria and, therefore, be organized in an index structure. To illustrate the usefulness of the *signature*, we consider a pattern-matching process where the signatures of a set of patterns can be used as search keys of an index structure to return only possible layout's windows that should be analyzed in a *hotspot*-detection process.

This work presents a *signature* for 2D spatial configurations as a pair of vectors of numbers representing changes between geometries and empty-spaces along the horizontal and vertical slices of a configuration. As the *signature* should be general enough to consider different possible patterns, we consider the representation of *range patterns* as in [6], which enables a compactly representation of similar exact patterns. Using this representation, a method extracts signatures from *range patterns*, and then use these signatures as search criteria over the inverted index to retrieve candidate windows that can match the patterns. The main difference with respect to our previous work is that the candidate searching can be done even if the layout's windows from where the *signature* are extracted are not of the same size of the pattern realization. Here, a pattern realization is understood as a specific geometrical configuration in the space that a *range pattern* specification can generate. Also, as additional contribution, a detailed analysis is executed for each element involved in the implementation of the theoretical work (e.g., extraction method, data storing structure, and so on) comparing different decisions.

In summary, the key contributions presented in this chapter are:

- Definition of a *signature* for configurations composed of non-overlapping rectangles.
- Implementation of algorithms for the extraction of the *signature* from layouts by windows and from *range patterns*.
- Use of the *signature* with an indexing structure to efficiently search for candidate windows that can match a *hotspot* pattern.

3.1 Signature.

As we have said, the filtering-out of layout portions is based on the use of a *signature*. What is a signature? In this work, a *signature* is a mapping from a geometric configuration composed of non-overlapping rectangles to one or more numbers. Unlike signatures characterizing single shapes such as those in [17] and [21], a *signature* here has to characterize a set of shapes (rectangles). This set of shapes will represent a set of IC components that in a design appear as rectangular shapes. So, a simple but effective *signature* that characterizes two-dimensional configurations composed of non-overlapping rectangles located along or perpendicular to $x-$ or $y-$axis is introduced.

To formalize this *signature*, we first define the concepts of grid and slices of a representation of two-dimensional configurations.

Definition 1 *A $N \times M$ regular grid representation of a configuration of rectangular shapes (e.g. a set of IC components) is composed by $N \times M$ of cells of the same size in a two-dimensional space, such that there exist N horizontal units and for each of these N units, there exist M cells along the horizontal axis.*

The granularity of this representation is such that a cell of the grid overlaps a rectangle (and have a value of 1), or it do not overlaps a rectangle at all (and have a value of 0). Simplifying the concept, we can say that a regular grid representation of a configuration is a binary-cell matrix, where all the cells are squares of the same size (i.e. the width and the height of each cell is the same), and correspond to the design resolution (i.e. the minimum printable element).

Additionally to the regular grid representation, we have an irregular grid representation. This irregular grid representation is based on grid slices, which are defined as follows:

Definition 2 *Horizontal(vertical) slice of a regular grid representation is a set of contiguous horizontal(vertical) cells (rows or columns, respectively) of the regular grid that are equal. Each slice contains a number of fragments that group equal cells in the orthogonal axis (a fragment can be conceived as a sub-matrix where all the cells have the same value).*

With the previous defined concepts, we can define the last concept called cutting-slice.

Definition 3 *A cutting-slice is a set of horizontal or vertical slices* $\{S_0, \ldots, S_{n-1}\}$ *that meets the following specifications:*

1. *Adjacent slices are not equal, i.e.* $S_i \neq S_{i+1}$, $0 \leq i \leq (n-2)$
2. *Each slice* S_i *is decomposed into fragments* $\{F_{i,0}, \ldots, F_{i,m-1}\}$*, where* $F_{i,j} \neq F_{i,j+1}$, $0 \leq j \leq (m-2)$

Thus, the irregular grid representation is defined as the representation formed by the pair of horizontal and vertical cutting-slices of a regular grid representation. Figure 3.1 shows and example of the different representations of a simple configuration.

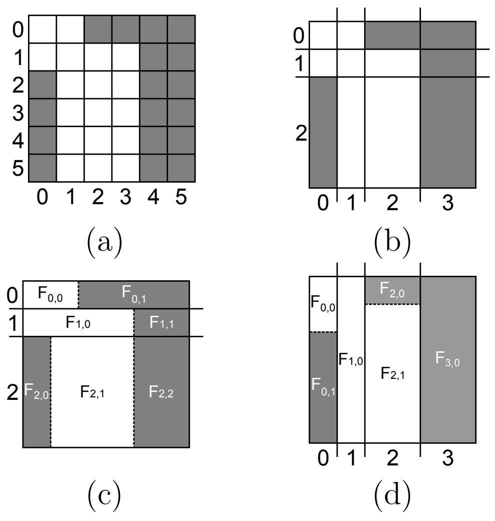


Figure 3.1: Regular and irregular grid representation.

In figure 3.1(a) we can see a regular representation of a 6×6 simple configuration. There, all the cells are squares of the same size, and the size of the each cell is the minimum size (length or width) of a shape. Figure 3.1(b) shows the irregular representation of the same configuration. Here, we can notice that the 4^{th} and 5^{th} columns in figure 3.1(a) are merged into the 3^{rd} column in figure 3.1(b) because both columns are equal. The same occurs to the 2^{nd} to 5^{th} rows in figure 3.1(a), that are merged into the 2^{nd} row in figure 3.1(b). Note that, if not all the rows from the 2^{nd} to 5^{th} in figure 3.1(a) were merged into a single one row, we would have two horizontal slices in the irregular grid representation that would be equal, and would not satisfy the cutting-slice specifications.

Using the previous definitions, we can proceed to define the *signature* as it follows:

Definition 4 *Given and $N \times M$ grid representation of a two-dimensional configuration of non-overlapping rectangles, the change-based signature ($\mathcal{S}^+$) is a tuple of the form ($[h_0, \ldots, h_N]$, $[v_0, \ldots, v_M]$), where h_i ($0 \leq i \leq N$) is the number of changes of values along the horizontal slice i and v_j ($0 \leq j \leq M$) is the number of changes of values along the vertical slice j.*

Here, a change of value ocurrs when, given a cell in a slice, the adjacent cell has the opposite value (i.e., 0 to 1, or 1 to 0). For example, in the slice $[1, 0, 1]$ we have 2 changes of values; 1 to 0 (0^{th} to 1^{th} position) and then 0 to 1 (1^{th} to 2^{nd} position).

Notice that the definition of $\mathcal{S}^+$ applies to grid representation, and therefore, to regular and irregular representations. In what follows, and unless the contrary is explicitly stated, $\mathcal{S}^+$ will refer to the *signature* over irregular grid representations.

Using figure 3.1(b) as an example, we can see that its $\mathcal{S}^+$ is the tuple ([1, 1, 2], [1, 0, 1, 0]). Figure 3.1(c) and (d) shows us the horizontal and vertical slices, respectively, each of them with its corresponding fragments. As an example, $F_{0,0}$ corresponds to the fragment 0 of the 0^{th} horizontal slice, which corresponds to a two-cell block in the corresponding regular grid representation (the cell at (0,0) and the cell at (0,1) in figure 3.1(a)).

It is important to note that $\mathcal{S}^+$ has the following properties:

- $\mathcal{S}^+$ is scale independent when extracted from an irregular grid representation. It is easy to see that $\mathcal{S}^+$ will not change as we apply a continuous scaling over the whole configuration, because equal rows or columns are represented by a single slice. Even more, $\mathcal{S}^+$ will be the same for patterns that are similar but differ in the distance and the width of rectangles. As an example, if the width of the 1^{th} vertical slice in figure 3.2(d) was wider, the vertical part of the $\mathcal{S}^+$ tuple would even be [1, 0, 1, 0].

- The number of changes along a slice is equivalent to the number of fragments in the slice minus 1.

3.2 Design layout representation and *signature* extraction.

A design layout is a computational representation of a design, which specifies the organization of electronic components of an IC. In Chapter 2, we have seen that, for manufacturing purposes, what really matters is to produce a mask from a design. From now, we will refer to this mask as the layout, or what the layout represents, i.e., a set of geometrical elements. These representations are typically stored in the industry as GDSII format files that are constructed using CAD tools and could represent a chip area range from a few square millimeters to around 450 mm^2 using from few GBs up to 100GB of space.

In this work, and similarly to [6], a layout is represented by a $N \times M$ regular grid L, denoted by $L_{N \times M}$, where N and M depend on the granularity of the layout representation, which is typically equal to the manufacturing grid size. This manufacturing grid size depends on the manufacturing technology, and decreases each year until the current 25nm technology. Each element $L[i, j]$, with $0 \leq i \leq N$ and $0 \leq j \leq M$ is associated with a spatial location in the manufacturing grid, such that $L[i, j] = 1$ if there exists a rectangle that overlaps this location, and $L[i, j] = 0$, otherwise.

Figure 3.2 shows a simple example of a layout representation using the same configuration used previously. Similar to what occurs in Chapter 3.1, the granularity of the grid is such that each cell represents the minimum size of any unit in the layout, so that a cell overlaps a rectangle or it does not overlap any rectangle at all.

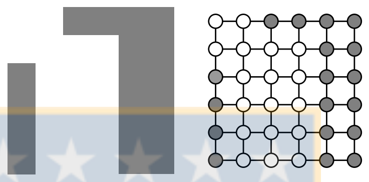


Figure 3.2: Example of a layout representation

The $N \times M$ regular grid L representing a layout is pre-processed by windows in an off-line process applied previously to the *hotspot* detection step. These windows are then mapped to irregular grids to extract $\mathcal{S}^+$. A window corresponds to a rectangular sub-portion of the layout and is defined as a $N' \times M'$ sub-grid W of L, denoted by $W_{N' \times M'}$, with $1 \leq N' \leq N$ and $1 \leq M' \leq M$, and where rows (and columns) are consecutive in L.

The *signature* extraction process starts with a window located at the top-left corner of the layout grid. Without changing the window dimensions, the window slides one-by-one position along the $x-$axis and then along the $y-$axis. Thus, given a layout grid of size $N \times M$ and a window size of $N' \times M'$, the number of windows in the layout is $(N – N' + 1) \times (M – M' + 1)$.

Windows are identified by using a correlative number following the window sliding. There exists a direct mapping from this ID to the initial coordinates (top-left corner) of the window in the layout. Given a layout $L_{N \times M}$, windows of size $N' \times M'$, and a number i representing the correlative window incremented by sliding the window horizontally and then vertically, coordinates for the top-left corner of the window are $(i/(M–M'+1),\ i\ mod(M–M'+1))$. Conversely, given a coordinate pair (x, y) for the top-left corner of a window, its ID i is $(y \times (M–M'+1)) + x$.

For each window, the horizontal and vertical slices are determined and $\mathcal{S}^+$ is extracted from the irregular-grid representation of the window by using the following algorithm.

Algorithm 1 Generates the $\mathcal{S}^+$ of a layout window.

```
    Input $layout$ : A layout in binary grid representation, $x, y$ : (x, y) coordinates of the window, $width$ : window's width, $height$ : window's height.
    Output $signature$ : The $signature$ of the window as a pair of lists: ($horizontal\_changes, vertical\_changes$).
 1: $signature \leftarrow$ empty pair
 2: $horizontal\_signature \leftarrow$ empty list
 3: for $j = 0$ to ($height$-1) do
 4:     $equal\_next\_row \leftarrow$ TRUE
 5:     $changes \leftarrow 0$
 6:     $last\_value \leftarrow layout[y + j][x]$
 7:     for $i = 0$ to ($width$-1) do
 8:         if $layout[y + j][x + i] \neq last\_value$ then
 9:             $changes \leftarrow changes$+1
10:             $last\_value \leftarrow layout[y + j][x + i]$
11:         if $layout[y + j][x + i] \neq layout[y + j + 1][x + i]$ then
12:             $equal\_next\_row \leftarrow$ FALSE
13:     if (not $equal\_next\_row$) $\vee$ ($j = (height$-1)) then
14:         $horizontal\_signature \leftarrow$ Append $changes$
15: $vertical\_signature \leftarrow$ empty list
16: for $i = 0$ to ($width$-1) do
17:     $equal\_next\_column \leftarrow$ TRUE
18:     $changes \leftarrow 0$
19:     $last\_value \leftarrow layout[y][x + i]$
20:     for $j = 0$ to ($height$-1) do
21:         if $layout[y + j][x + i] \neq last\_value$ then
22:             $changes \leftarrow changes$+1
23:             $last\_value \leftarrow layout[y + j][x + i]$
24:         if $layout[y + j][x + i] \neq layout[y + j][x + i + 1]$ then
25:             $equal\_next\_column \leftarrow$ FALSE
26:     if (not $equal\_next\_column$) $\vee$ ($i = (width$-1)) then
27:         $vertical\_signature \leftarrow$ Append $changes$
28: $signature.first \leftarrow horizontal\_signature$
29: $signature.second \leftarrow vertical\_signature$
30: return $signature$
```

Let us consider the simple layout represented with a 8×8 regular grid shown in figure 3.3.

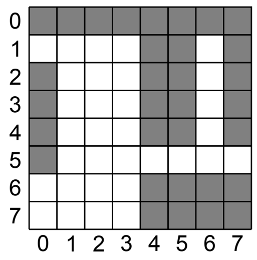


Figure 3.3: Example of a simple layout

Assume now a window size of 6×6, then the number of windows within the layout is 9. Figures 3.4(a)-(c) show windows created along the three possible horizontal rows. These windows are created using the movement explained previously.

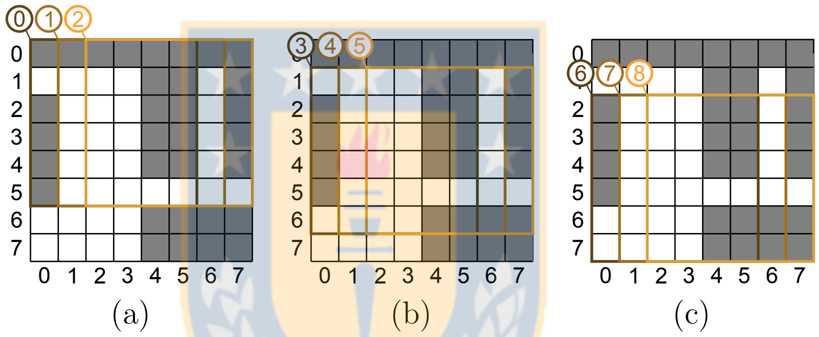
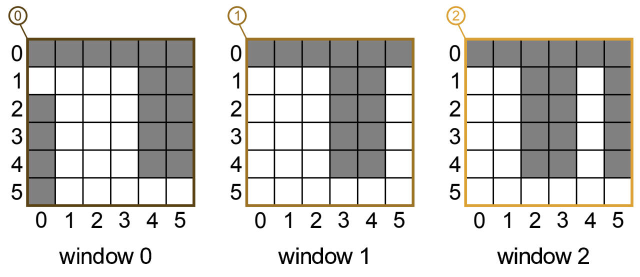


Figure 3.4: Example of window creation.

As figures 3.5(a)-(c) show, $\mathcal{S}^+$ for window 0 is ([0, 1, 2, 1], [2, 1, 1]), for window 1 is ([0, 2, 0], [1, 1, 1]) and for window 2 is ([0, 3, 0], [1, 1, 1, 1]). It is important to notice that a large layout can be partitioned to make a distributed extraction of $\mathcal{S}^+$. This is a domain decomposition processing of a layout. An important consideration is that consecutive portions of the layout sent to different nodes should overlap to be able to detect patterns located at their intersection.



(a) (b) (c)

Figure 3.5: Example of *signature* extraction for different windows.

Suppose a layout of $m \times n$, and a size of window extraction of $i \times j$. Because, for each cell in the horizontal orientation (except the last $i - 1$ cells) we calculate the $\mathcal{S}^+$ for a window of width i (we must iterate over i cells horizontally), we have $m \times i$ cell calculation in the horizontal orientation. Similarly, for each cell in the vertical orientation (except the last $j - 1$ cells) we calculate the $\mathcal{S}^+$ for a window of height j (we must iterate over j cells vertically). Assuming the checking changes on slices has a constant time cost, the cost of the algorithm 1 is $O(i \times m) \times O(j \times n)$. Given that i and j are constants clearly smaller than m and n, respectively, the cost of the algorithm is $O(m \times n)$.

3.3 Pattern representation and *signature* extraction.

Patterns are two-dimensional configurations composed of non-overlapping rectangles. As such, they can also be described as irregular grids. Patterns of the same size can have small differences such that one could say that there exists a representative pattern with several similar realizations that vary in the distance between rectangles' boundaries. To avoid to represent each particular occurrence of a representative pattern (a realization), it is possible to compact geometric information of patterns by using the specification of *range patterns*. As the work in [6] presents, a *range pattern* is a configuration of two-dimensional non-overlapping rectangles with additional specifications about the horizontal and vertical distances between rectangles's boundaries as restrictions. The figures 3.6(a)-(c) illustrate the case of three different realizations of a general pattern that is then codified as a *range pattern*. All of these three realizations are composed of three rectangles with different distances between boundaries.

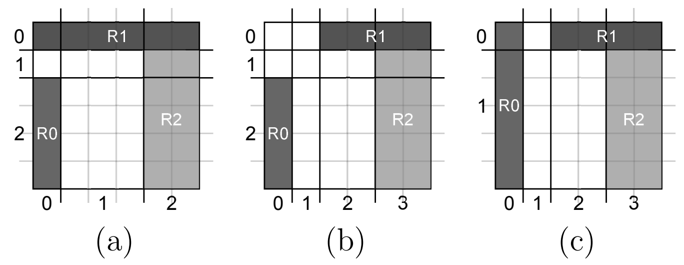


Figure 3.6: Three different *realizations* of the same representative pattern.

It is possible to compress the specification of these three patterns by using a *range pattern* specified as figure 3.7 shows.

Horizontal constraints	Vertical constraints
1. $R_0.R - R_0.L$ is 1	1. $R_0.T - R_0.B = (4,6)$
2. $R_1.R - R_1.L = (4,6)$	2. $R_1.T - R_1.B$ is 1
3. $R_2.R - R_2.L$ is 2	3. $R_2.T - R_2.B$ is 5
4. $R_2.L - R_0.R$ is 3	4. $R_1.B - R_2.T$ is 0
5. $R_2.R - R_1.R$ is 0	5. $R_0.B - R_2.B$ is 0

Figure 3.7: A *range pattern* specification.

$R_i.L$: left boundary of R_i, $R_i.R$: right boundary, $R_i.T$: top boundary, $R_i.B$: bottom boundary.

Since a *range pattern* may have several realizations, it is not possible to extract $\mathcal{S}^+$ from a *range pattern* directly. $\mathcal{S}^+$ uses the horizontal and vertical slices of a realization of the pattern. Consequently, it is necessary to derive possible realizations from *range patterns* that have the same irregular grid representation, i.e., the same horizontal and vertical slices. This is done by first mapping the *range pattern* specification to horizontal and vertical *range graphs* independently (HRG and VRG, respectively). These *range graphs* are then combined to derive possible irregular grid representations.

A *range graph* is introduced in [6], and it is formally defined as follows:

Definition 5 *A range graph G is a quadruple (V, E, ψ, ω) where V and E are finite sets, $\psi : E \to \{(v, w) \in V \times V; v \neq w\}$ and $\omega : E \to \{(m, n) \in R \times R; m \leq n\}$.*

The elements of V are vertexes, the elements of E are edges and the elements of R are real numbers. G satisfies the condition that whenever there is an edge

$e = (v, w) \in E(G)$, with $\omega(e) = (m, n)$, there is also an edge $\tilde{e} \in E(G)$ where $\tilde{e} = (w, v)$ and $\omega(\tilde{e}) = (-n, -m)$. Here, m (n) is denoted as $min(e)$ ($max(e)$) and is called the lower (upper) bound of the edge e. The *range* of e is $|\omega(e)| = (n–m)$.

A *range graph* (V, E, ψ, ω) is called *stable* if and only if the following condition is satisfied: for each edge $e \in E(G)$, *range* of e is finite and minimized. A *range* of e is finite if and only if $|\omega(e)| = (n–m) \neq \infty$.

To understand the use of a *range graph*, we can say that in a *range graph* G:

- Its set of vertexes V are the boundaries of the rectangles in the same direction of the *range graph*.
- Its set of edges E connects all the different boundaries.
- The lower and upper bounds of an edge $e \in E$ specify the distance between the corresponding boundaries of the edge e, taking as reference the left border or the bottom border of the pattern.

Figure 3.8 shows an example of distances between rectangles' boundaries. In figure 3.8(a), the distance between $R0.r$ and $R1.l$ is 1, and the distance between $R0.b$ and $R1.t$ is 1. In figure 3.8(b), the distance between $R0.r$ and $R1.l$ is -1, while in figure 3.8(c), the distance between $R0.b$ and $R1.t$ is 0.

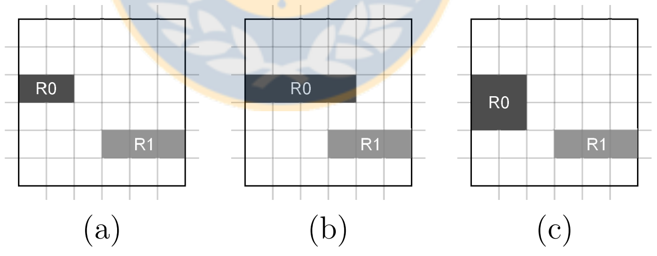


Figure 3.8: Example of rectangles' boundaries distances.

As we can see in the previous example, the distance between two rectangles's boundaries can lead us to different situations. One situation occurs when the first rectangle's boundary appears before the second rectangle's boundary. The second one occurs when the first rectangle's boundary appears aligned with the second rectangle's boundary, and the third situation occurs when the first rectangle's boundary appears after the second rectangle's boundary. In the case that a restriction in the *range pattern* allows one, two or three of these situations to occur, then one, two or three slices will be produced for these rectangles's boundaries interaction.

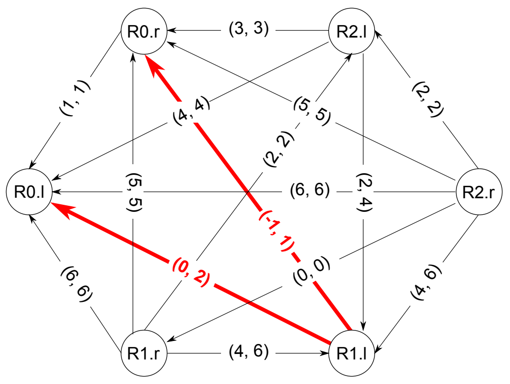
An edge $e \in E(G)$ where $\omega(e) = (m, n)$ is called *indefinite* if and only if $m \neq n, m \leq 0$ and $n \geq 0$. Otherwise, e is called *definite*. An *indefinite* edge $e \in E(G)$ with $\omega(e) = (m, n)$ contains a set of *definite* ranges:

1. If $m < 0$ and $n > 0$, then there are three *definite* ranges: $\{(m, -1), (0, 0), (1, n)\}$.
2. If $m < 0$ and $n = 0$, then there are two *definite* ranges: $\{(m, -1), (0, 0)\}$.
3. If $m = 0$ and $n > 0$, there are two *definite* ranges: $\{(0, 0), (1, n)\}$.

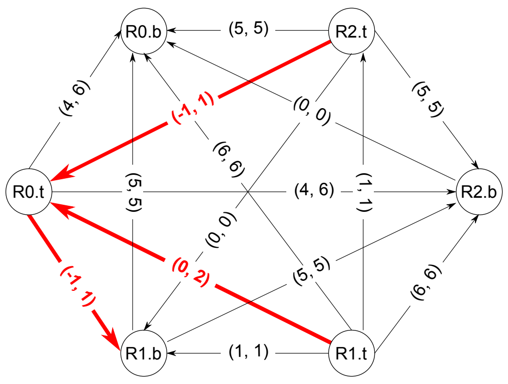
Definition 6 *A stable range graph $G = (V, E, \psi, \omega)$ is called definite if and only if each edge $e \in E$ is definite.*

As we have said, the specification of a *range pattern* is converted to two *range graph*: one for the horizontal boundaries of the rectangles and another for the vertical boundaries of the rectangles. Thus, each rectangle's boundary r_i becomes a vertex v_i in the corresponding *range graph* and an edge e_{ij} exists in the *range graph* between any two vertexes v_i and v_j, where $i \neq j$, with the lower and upper bounds given in the *range pattern* specification. If there is not a restriction, $\omega(e_{ij})$ is set to $(-\infty, \infty)$.

Figure 3.9 shows the horizontal and vertical *range graphs* from the specification of the *range pattern* in figure 3.7. In this figure, edges in red denote edges that are *indefinite* and that lead to multiple *definite* edges.



(a)



(b)

Figure 3.9: *Range graphs* of the *range pattern* in Figure 3.7.

(a) horizontal *range graph* and (b) vertical *range graph*.

Since the distance relationship between two rectangles's boundaries can be affected by the distance relationships that they have with other edges, we need to update the distance relationships between each pair of rectangles's boundaries, i.e., the lower and upper bounds of each edge of the graph. This update is done from the more restrictive point of view, i.e., given three rectangles's boundaries, that relationship being the most restrictive will remains, while the second relationship will be updated as the first relationship indicates. As an example, suppose the four restrictions in figure 3.10.

1. $R_0.R - R_0.L$ is 2
2. $R_1.R - R_1.L$ is $(1, 2)$
3. $R_0.L - R_1.R$ is 0
4. $R_0.R - R_1.L$ is 3

Figure 3.10: Four distance restrictions.

The first restriction ($R_0.R - R_0.L$ is 2) indicates that the width of $R0$ is 2. The second restriction ($R_1.R - R_1.L$ is $(1, 2)$) indicates that the width of $R1$ could be 1 or 2. The third restriction $R_0.L - R_1.R$ is 0 indicates that the distance between $R0.L$ and $R1.R$ is zero. In other words, just $R0$ finishes, $R1$ starts. The fourth restriction ($R_0.R - R_1.L$ is 3) indicates that the distance between the start of $R0$ and the end of $R1$ is 3. This restriction affects the width of $R1$, limiting it to 1, not 1 or 2 anymore.

The algorithm that does this update is the **All-Pair Min-Range Path (APMRP)** algorithm. This algorithm runs over all of the rectangles's boundaries (nodes of a *range graph*), updating all of its relationships with the others nodes (the edges's boundaries associated with that node) in a three-level nested loop, reason why the cost of the APMRP algorithm is $O(n^3)$, being n the number of nodes (rectangles's boundaries) in the corresponding *range graph.* The APMRP algorithm works as follows:

Algorithm 2 APMRP algorithm.

```
   Input A range graph $G = (V, E, \psi, \omega)$
   Output A range graph $G'$
1: for $k = 0$ to $(|V| - 1)$ do
2:     for $i = 0$ to $(|V| - 1)$ do
3:         for $j = 0$ to $(|V| - 1)$ do
4:             if $(i \neq j)$ and $(j \neq k)$ and $(i \neq k)$ then
5:                 if $(max(e_{ik}) < \infty)$ and $(max(e_{kj}) < \infty)$ and $\{max(e_{ik}) + max(e_{kj}) < max(e_{ij})\}$ then
6:                     $max(e_{ij}) \leftarrow max(e_{ik}) + max(e_{kj})$
7:                 if $(min(e_{ik}) > -\infty)$ and $(min(e_{kj}) > -\infty)$ and $\{min(e_{ik}) + min(e_{kj}) > min(e_{ij})\}$ then
8:                     $min(e_{ij}) \leftarrow min(e_{ik}) + min(e_{kj})$
   return G
```

When applying the APMRP algorithm to a *range graph* G, the lower bounds of some edges can increase and the upper bounds of some edges can decrease. If the *range* of an edge $e \in E(G)$ becomes negative or unbounded after the application of the APMRP algorithm, it can be concluded that the specification for the *range pattern* is invalid and needs to be revised. In all other cases, the APMRP algorithm finds a *stable range graph* G' for the given input *range graph* G. Also, it is not necessary that a *definite stable graph* is obtained after the application of the APMRP algorithm. Given an *indefinite stable graph* G^s, it is necessary to convert the *indefinite* edges of G^s into *definite* ones to determine the unique topological orders of the rectangles's edges, because each topological order corresponds to a cutting-slice.

The **Enumerate Definite Range-graphs (ENUM_DRG)** algorithm takes an *indefinite stable graph* G^s and outputs all the *definite range graphs* contained in it. This algorithm works as follows:

This algorithm runs over all the edges of a *range graph.* When it finds an *indefinite* edge, it converts this *indefinite* edge into the 2 or 3 *definite* ones, depending on the original edge's boundaries, and then run the APMRP algorithm. As a *range graph* is a complete graph, if n is the number of nodes in the graph, then $n \times (n-1)/2$ is

Algorithm 3 ENUM_DRG algorithm.

```
    Input An indefinite range graph $G^s = (V, E, \psi, \omega)$
    Output All definite range graphs contained in $G^s$
 1: Invoke APMRP($G^S$) to update the ranges of each edge in $G^s$
 2: $allEdgeDefinite \leftarrow TRUE$
 3: for each edge $e = (v, w) \in E(G^s)$ do
 4:     if $e$ is indefinite then
 5:         for each definite range $r = (min, max)$ of $e$ do
 6:             $\omega(e(v, w)) \leftarrow (min, max)$
 7:             $\omega(\tilde{e}(w, v)) \leftarrow (-max, -min)$
 8:             Invoke ENUM_DRG on the modified $G^s$
 9:         $allEdgeDefinite \leftarrow FALSE$
10:     if $allEdgeDefinite = FALSE$ then
11:         Break
12: if $allEdgeDefinite = TRUE$ then
13:     Output definite range graph $G^d$
```

the number of edges in the graph. For this reason, the cost of the ENUM_DRG is $O(n^2) \times O(n^3)$, i.e., $O(n^5)$, being n the number of nodes of the *range graph*.

After the execution of the ENUM_DRG algorithm, we obtain all the *definite range graphs* contained in an *indefinite range graph*. Thus, the ENUM_DRG algorithm is invoked over the HRG and the VRG to obtain all the cutting-slices needed to represent a *range graph*. The cutting-slices needed to represent a *range graph* can be derived from the *definite stable range graphs* obtained using the ENUM_DRG algorithm by using the *precedence* and *equivalence* relations.

Given a *definite* edge $e = (v, w) \in E(G)$, where $\omega(e) = (m, n)$, vertex v is said to precede vertex w if and only if $m > 0$. Two vertexes are equal if and only if $m = n = 0$. If vertex v precedes vertex w, then vertex v and vertex w are said to satisfy the precedence relation R_p denoted vR_pw. If vertex v equals vertex w, then vertex v and vertex w are said to satisfy the equivalence relation denoted as vR_ew. Since, a *definite range graph* is complete and contains only *definite* edges by definition, it is easy to see that in a *definite range graph* $G^d = (V, E, \psi, \omega)$ any pair of vertexes v and $w \in V(G)(v \neq w)$ satisfies one of the following three conditions: (1) vR_pw; (2) wR_pv; (3) vR_ew and wR_ev. Note that the topological order of the vertexes can be derived according to the precedence and equivalence relation between vertexes. Topological orders are specified by a sequence of sub lists sorted by coordinates along an axis. Each sub list, called *limit*, is the list of boundaries that share the same coordinate, i.e., satisfy the equivalence relation in the corresponding *definite range graph*. Intuitively a *limit* defines the start or end interval of a slice in the corresponding axis and the equivalence relationship establishes that the edges of rectangles are the same when projected onto the corresponding axis. For example, consider figure 3.11 and the horizontal topological order. This topological order is represented as a list of sub lists of the form $[[R0.l, R1.l], [R1.r], [R0.l, R2.l], [R2.r]]$, where $Ri.l$ and $Ri.r$ indicate the left and right edges of a rectangle i, respectively. Each sub list (e.g., $[R0.l, R1.l]$) is a *limit* in the topological order.

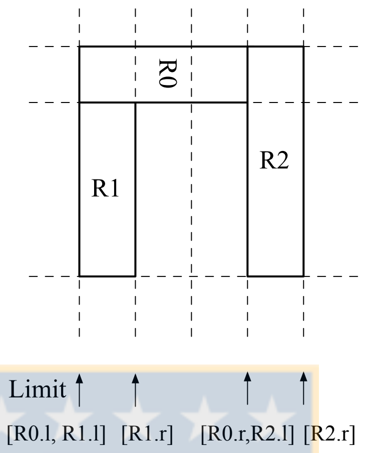
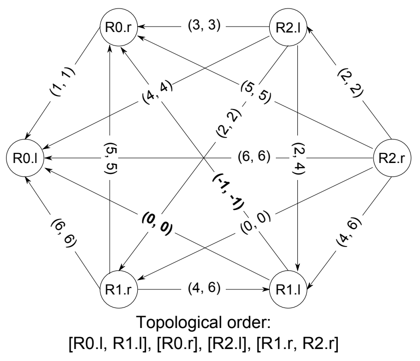
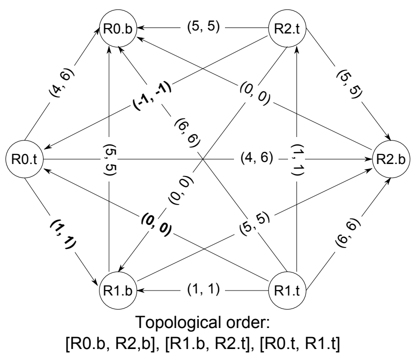


Figure 3.11: Example of an horizontal topological order.

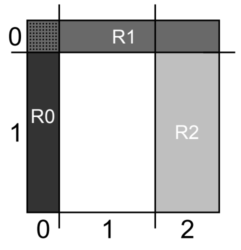
As a *definite range graph* defines a topological order in which limits of slices can be defined, and each topological order corresponds to a cutting-slice, we can derive $\mathcal{S}^+$ using both horizontal and vertical *definite range graphs*. In these graphs, each label on an edge defines a possible range of distance between boundaries, where the range is of the form (i, j) or $(0, 0)$, with i and j being both positive or negative. By using the previous mentioned algorithms, we extract *definite* vertical and horizontal *range graphs*. But this is not enough since combinations of graphs may produce inconsistent configuration, that is, configurations where rectangles overlap. As an example, we can take the *range graphs* of figure 3.9. They derive 3 *definite* horizontal and 3 *definite* vertical graphs. The combination of these graphs produces initially 9 realizations, where one of them is inconsistent. Figure 3.12(a)-(b) shows the horizontal and vertical topological orders that derive the corresponding inconsistent realization shown in figure 3.12(c), where R_0 and R_1 overlap in the dotted area.



(a)



(b)



(c)

Figure 3.12: *Definite* HRG and VRG that derive an incompatible realization.

(a) *Definite* HRG, (b) *definite* VRG and (c) inconsistent resulting configuration.

To overcome this situation, the algorithm GET_SIG (see algorithm 3.3) combines both topological orders and checks if the combination is consistent. If it is so, the algorithm returns the $\mathcal{S}^+$ of the realization derived from the given topological orders. For each edge, variables *rectangle* and *type* indicate the rectangle to which the edge belongs and the type of the edge (i.e., left, right, top, or bottom). For example, for *limit* $L = [R1.r]$ and edge $e = R1.r$, $e.rectangle = 1$ and $e.type$ is *right*. Finally, rectangles are numerated as consecutive integers.

Algorithm 4 Generates the $\mathcal{S}^+$ on two topological orders.

```
    Input $primary\_topological\_order$: Topological order array, $secondary\_topological\_order$: Topological order array
    Output $\mathcal{S}^+$ derived from the two topological orders
 1: $signature \leftarrow$ empty pair
 2: $status \leftarrow$ empty list
 3: $changes\_signature \leftarrow$ empty list
 4: $open\_rectangles \leftarrow 0$
 5: for all rectangle $i$ in $primary\_topological\_order$ do
 6:     $status[i] \leftarrow 0$
 7: for all limit $L$ in $primary\_topological\_order$ do
 8:     for all edge $e$ in $L$ do
 9:         if $(e.type = \text{left}) \vee (e.type = \text{bottom})$ then
10:             $status[e.rectangle] \leftarrow 1$
11:             $open\_rectangles \leftarrow open\_rectangles + 1$
12:         else if $(e.type = \text{right}) \vee (e.type = \text{top})$ then
13:             $status[e.rectangle] \leftarrow 0$
14:             $open\_rectangles \leftarrow open\_rectangles - 1$
15:     $changes \leftarrow 0$
16:     if $open\_rectangles > 0$ then
17:         $ortogonal\_open\_rectangles \leftarrow 0$
18:         for all limit $L'$ in $secondary\_topological\_order$ do
19:             for all edge $e'$ in $L'$ do
20:                 if $((e'.type = \text{bottom}) \vee (e'.type = \text{left})) \wedge status[e'.rectangle] = 1$ then
21:                     if $(L'.index \neq 0) \wedge (L'.index \neq (|secondary\_topological\_order| - 1))$ then
22:                         if $|L'| > 1$ then
23:                             $sum \leftarrow 0$
24:                             for all edge $e$ in $L'$ do
25:                                 $sum \leftarrow sum + status[e.rectangle]$
26:                             if $sum < 2$ then
27:                                 $changes \leftarrow changes + 1$
28:                         else
29:                             $changes \leftarrow changes + 1$
30:                     $ortogonal\_open\_rectangles \leftarrow ortogonal\_open\_rectangles + 1$
31:                 else if $((e'.type = \text{top}) \vee (e'.type = \text{right})) \wedge status[e'.rectangle] = 1$ then
32:                     if $(L'.index \neq 0) \wedge (L'.index \neq (|secondary\_topological\_order| - 1))$ then
33:                         if $|L'| > 0$ then
34:                             $sum \leftarrow 0$
35:                             for all edge $e$ in $L'$ do
36:                                 $sum \leftarrow sum + status[e.rectangle]$
37:                             if $sum < 2$ then
38:                                 $changes \leftarrow changes + 1$
39:                         else
40:                             $changes \leftarrow changes + 1$
41:                     $ortogonal\_open\_rectangles \leftarrow ortogonal\_open\_rectangles - 1$
42:                 if $(ortogonal\_open\_rectangles > 1) \wedge (e'.index = (|L'| - 1))$ then
43:                     return $Error$
44:         $changes\_signature \leftarrow$ Append $changes$
45:     else
46:         if $L.index \neq |primary\_topological\_order| - 1$ then
47:             $changes\_signature \leftarrow$ Append $changes$
48: $signature.first \leftarrow changes\_signature$*
49: $signature.second \leftarrow GET\_SIG(secondary\_topological\_order, primary\_topological\_order)$*
50: return $signature$
```

**(ocurrs only one time)*

The algorithm checks whether or not the combination of an horizontal (HTO) and a vertical (VTO) topological order, derived from the horizontal and vertical *range graphs*, is consistent. A combination of topological orders is said to be consistent if their rectangles do not overlap in the space. If a combination is consistent, the algorithm returns its $\mathcal{S}^+$, otherwise, it returns $Error$. The basic idea of the algorithm is to check if rectangles overlap. To do so, the algorithm goes from left to right checking the HTO and from bottom to top checking the VTO. HTO or VTO can be either the primary or secondary topological order. If the algorithm finds a left edge of a rectangle in going through the HTO, it means that the rectangles is open, that is, the rectangle starts to appear as the algorithm goes over the list of limits in the HTO. Similarly if the rectangle finds a bottom edge in going through the VTO, the rectangle starts to appear as the algorithm goes over the list of limits in the VTO. Rectangles are closed if their right or top edges are found when going through the HTO or VTO, respectively. Then, rectangles overlap if by combining edges in the limits of HTO and VTO, they are open at the same time.

The algorithm handles a status variable (i.e., open or closed) for each rectangle that appears in a topological order. Note that if a rectangle appears in a vertical topological order, it must appear in an horizontal topological order, and vice versa. Initially all rectangles are closed (lines 5-6). For each limit L in the primary topological order (line 7), the algorithm analyzes each edge (lines 8-14). If the edge is a left or a bottom edge for HTO or VTO, respectively, the rectangle associated with the edge is open. Otherwise, the rectangle is closed and it will not appear in the next *limits* of the order. If there exist open rectangles in limit L of the primary topological order (line 16), the algorithm checks now each edge e' in each limit L' of the secondary topological order (lines 18-43). When analyzing the secondary topological order, the algorithm counts the number of rectangles that are open. Two cases are possible:

1. If the edge is a left or bottom, then the rectangle is open and the counter of open_orthogonal_rectangles is increased by 1 (lines 20-30).

2. If the edge is a right or top, then the rectangle is closed and the counter of open_orthogonal_rectangles is decreased by 1. (lines 31-41)

If after checking the edges of limit L', there exist open rectangles, the combination is not possible (line 42-43). This continues until all limits in HTO and VTO are analyzed. If after checking the HTO as primary topological order and the VTO as secondary topological order, there are no inconsistences, the algorithm proceeds to check the VTO as primary topological order and the HTO as secondary topological order (line 49). If there are no inconsistences in any of the two orientations (i.e., horizontally and vertically), then the algorithm returns the $\mathcal{S}^+$ of the pattern (line 50).

Let us suppose a range pattern definition with n rectangles. Both horizontal *range graph* and vertical *range graph* will have $2 \times n$ nodes, because each rectangle has two boundaries in each orientation (i.e., left and right boundary in the horizontal case, and top and bottom in the vertical case). This means that the corresponding topological orders will have $2 \times n$ elements. In algorithm , lines 5 and 6 go over the primary topological order, which means a cost $O(2 \times n) = O(n)$. From line 7 and line 8, we go over all the limits in the primary topological order. Each limit can contain at least 1 element, so the worst case is to have $2 \times n$ limits with 1 element in a topological order. For each element in the limit (line 8) we iterate over each limit and element in the secondary topological order (line 18 and line 19), which means $2 \times n$ nested iterations. At this point, we have a cost $O((2 \times n) \times (2 \times n)) + O(n) = O(4 \times n^2) + O(n) = O((4 \times n^2) + n) = O(n^2)$. If we assume that checking changes on slices has constant time cost, the algorithm is cost On^2 being n the number of rectangles in the pattern.

3.4 Signature-based searching.

Having patterns and layouts with their $\mathcal{S}^+$, it is possible to use $\mathcal{S}^+$ for selecting candidate windows of the layout that can contain a certain pattern (e.g., a *hotspot*). To do that, we use $\mathcal{S}^+$ as a search key in an inverted index. This inverted index is composed of a dictionary of possible values of $\mathcal{S}^+$ found in the layout. For each entry of the dictionary, there is a list where each element contains the ID of the window that matches the entry in the dictionary. Note that this structure can be created for a single layout or for several layouts if we consider to store not only the window's ID but also a layout's ID.

As an example, assume the *range pattern* of figure 3.7, which results in 8 different irregular grid realizations. Two of them have the same $\mathcal{S}^+$ (see figure 3.13(f) and (h)). The difference between these two realizations with the same $\mathcal{S}^+$ is the owner of the top-left corner (i.e., R_0 or R_1).

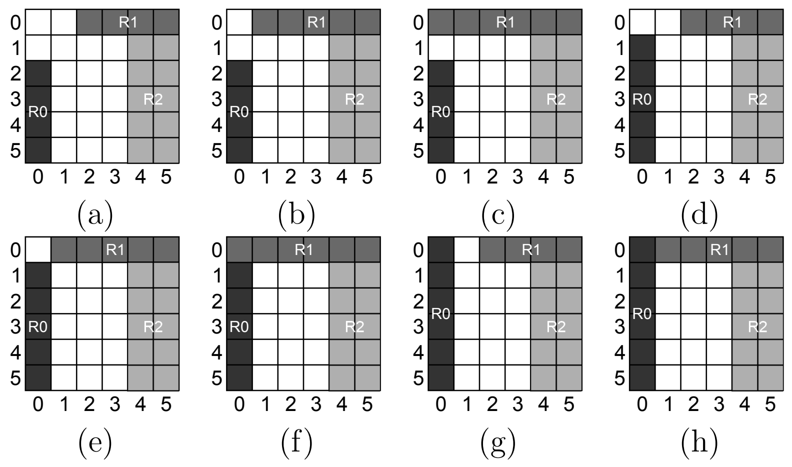


Figure 3.13: Different irregular grid realizations of *range pattern* in figure 3.7. (f) and (h) have the same $\mathcal{S}^+$.

Figure 3.14 shows the inverted index for the layout in figure 3.3. This layout contains 9 different windows (see figure 3.4), but 7 different $\mathcal{S}^+$. For the *range pattern* in figure 3.7, there are 7 different $\mathcal{S}^+$, and for each of them, there are 8 possible rotations, i.e., 0°, 90°, 180°, 270° and their mirror realizations. In total, searching for these 7 different $\mathcal{S}^+$ of the *range pattern* requires to search for $7 \times 8 = 56$ possible rotations, which requires the efficient use of indexing structures.

$\mathcal{S}^+$	**List of windows**
$([0,1,2,1],[2,1,1])$	$[0]$
$([0,2,0],[1,1,1])$	$[1]$
$([0,3,0],[1,1,1,1])$	$[2]$
$([1,2,1,1],[2,0,2])$	$[3]$
$([2,1,1],[1,0,2])$	$[6]$
$([2,0,1],[0,2,1])$	$[4,7]$
$([3,0,1],[0,2,1,2])$	$[5,8]$

Figure 3.14: Inverted index for layout in Figure 3.3

Searching for candidate windows takes the time needed for finding the *signature* in the dictionary and then, traversing the list of windows with the same $\mathcal{S}^+$. For efficiency, we can organize the dictionary with different types of structures such as a binary search tree, hash tables, and another more sophisticated structure.

One of the limitations of this approach is that it assumes that the window used to extract the signatures from the layout and the pattern are of the same size. If this condition does not occur, it is impossible to assert that equal $\mathcal{S}^+$ implies an equal rectangles configuration. As windows are processed before *range patterns*, and *range patterns* represent typically a potential *hotspot*, we can say that one must know the *range patterns* size before to the window extraction process, or, alternatively, one must have different inverted index structures for different window sizes and use the index that matches the *range pattern* size. The problem with this idea is that a *range pattern* could lead to different realizations, each one with different possible size. Even more, as *range patterns* represent a potential *hotspot*, and given that *hotspot* occurrences are related to different manufacturing processes, it is almost impossible to know beforehand what the size of a *hotspot* would be. Only when a *fab* specifies a certain technology of manufacture, one could know how and why a *hotspot* could appear, and consequently, specify a *range pattern* that could lead to potential *hotspots*.

There are two ways to address this limitation. The first one is to have windows larger enough to certainly know that any *range pattern* realization would be contained within a window. Then, a window segmentation procedure must be designed, because as $\mathcal{S}^+$ is derived from an irregular grid representation, it is not necessary true that a sub-portion of a window is represented by the same sub-portion of the slices of the whole window.

Take as an example the simple window shown in figure 3.15. In figure 3.15(a) we can see a window of 2×6 cells. Its irregular grid representation contains two horizontal slices, the first one with two fragments and the second one with one fragment. The horizontal $\mathcal{S}^+$ of this windows is $[1, 0]$. If we took a sub-window of 2×5 cells that starts at the 0^{th} cell of the window, we will have a sub-window like figure 3.15(b) shows. This sub-window has an irregular grid representation that contains only one horizontal slice with one fragment. The horizontal $\mathcal{S}^+$ of this sub-window is $[0]$. As we can see, if we take a sub-window of a window, some slices can merge, and the fragments division can change because the eliminated portion of the original window. This makes that the $\mathcal{S}^+$ calculation of a sub-window based on the original $\mathcal{S}^+$ is a non-trivial task.

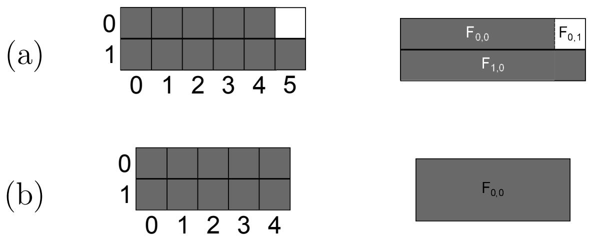


Figure 3.15: Example of a window and a sub-window

(a) a window with its irregular representation, (b) a sub-window with its irregular representation.

The second way to resolve the uncertainty of the correct window size is to have windows small enough to certainly know that no *range pattern* realization would be smaller than the window size. Using this idea, two approaches to match *range pattern* realizations could be used. The first one is to design a window-merging method to produce a window of the same size of a *range pattern* realization. The second one is to split the *range pattern* realization into portions that match the window size.

Using a window-merging method means that, every time that a pattern want to be detected, the window-merging method must be invoked to create windows that fit the pattern size. Even if the windows created by merging smaller windows were stored, the impractical time-cost (and memory-cost if the windows created were stored) makes this approach useless in practice. It makes more sense to split the pattern into several sub-patterns that cover the entire original pattern. These sub-patterns could be selected to fit the window size, and then use their $\mathcal{S}^+$ to retrieve candidate windows.

Given that the computational representation of a *range pattern* in our work is a *range graph*, we base the pattern splitting method in a *range graph* splitting method. First, we need to explicit the different sizes of a pattern realization. This can be achieved by traversing through the *range graphs*, converting the ranged-edges into its absolute edges. Here, a ranged-edge is understood as an edge whose lower bound and upper bound are different, while an absolute-edge is intended as an edge whose lower bound and upper bound are equal. We do this expliciting step because, as *range graphs* can origin different realization, each one with different dimensions, the splitting process could need different number of partitions for each realization. As example, the edge with $(1, 2)$ as lower and upper bounds is a ranged-edge, while the edges with $(1, 1)$ and $(2, 2)$ as lower and upper bounds, respectively, are absolute-edges. Note that, for a ranged-edge, it is not necessary to generate an edge for every possible value between the corresponding lower and upper bounds, but just for those that correspond to the regular grid granularity (i.e., the layout resolution).

Take as example an edge e, with lower and upper bounds of $100nm$ and $200nm$, respectively. Suppose a layout resolution of $50nm$. We only need to explicit the edges e_1 with lower bound of 100 and upper bound of 100, the edge e_2 with lower bound of 150 and upper bound of 150, and finally the edge e_3 with lower bound of 200 and upper bound of 200. This reduction in the number of edges to explicit is based in the fact that each grid cell represents the minimal printable element ($50nm$ in this example), and none intermediate value can be generated in the IC design. After one ranged-edge is converted into its absolute-edges, the APMRP algorithm must be invoked in order to update the nodes's relations in the *range graph*.

Having the *range graphs* with only absolute-edges, it is possible to know the exact distance between all pairs of graph nodes (i.e., rectangles's boundaries). We can traverse trough a *range graph* (i.e., horizontal or vertical *range graph*), counting the number of cells that exists between the right border of the pattern and the current position for the horizontal *range graph*, and the number of cells that exists between the bottom border of the pattern and the current position for the vertical *range graph*. When a number of cells equals to the size of the windows of extraction, all those nodes that were not reached yet in corresponding *range graph* are set at the same position of the last traversed node by setting all of its edges's values as 0 for the lower and upper bound. This means that, in a spatial configuration, all the rectangles's boundaries that are not reached are fixed in the window border. Then, another *range graph* starts with all the traversed nodes at the same position of the current node (lower and upper bounds fixed to 0), and the process starts again, until the last node of the *range graph* is reached.

Figure 3.16 shows a representation of the result of this process. There, figure 3.16(a) shows a pattern realization with its absolute graph. Figure 3.16(b), (c) and (d) show the divisions of the pattern, basing on the division of the corresponding absolute graph. Black nodes corresponds to nodes that were fixed.

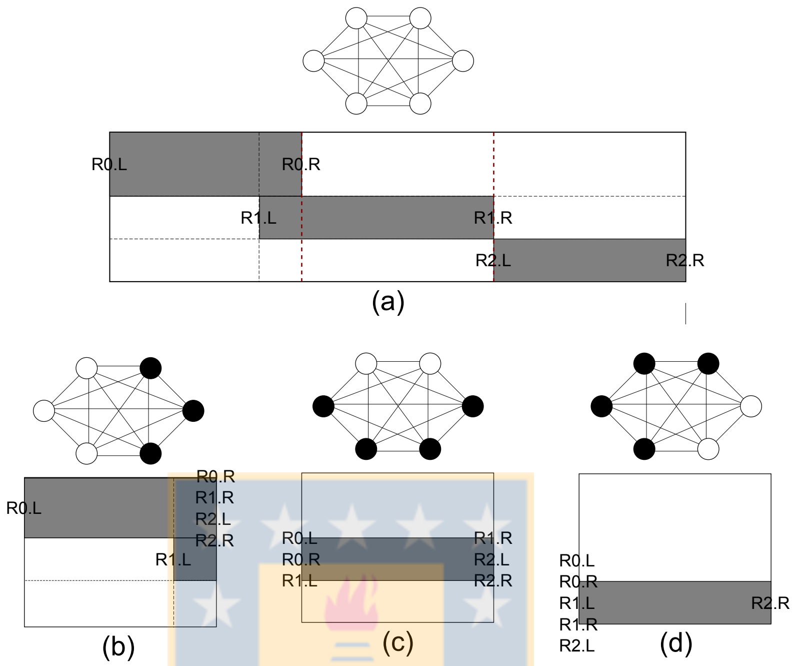


Figure 3.16: Example of the splitting process of a horizontal *range graph*.

(a) A *range graph* with its pattern size, (b) first split of the *range graph*, (c) second split of the *range graph*, (d) third split of the *range graph*.

The same process is performed for both orientations: horizontal *range graph* and vertical *range graph*. Then, a matrix of pairs of *range graph* partitions is processed to obtain topological orders, whose are then used as input for the GET_SIG algorithm. Let's say we obtain 3 different partitions from an absolute horizontal *range graph*: HRG_1, HRG_2, and HRG_3. Suppose we obtain 4 different partitions from an absolute vertical *range graph*: VRG_1, VRG_2, VRG_3, and VRG_4. The matrix of pairs of *range graphs* partitions is shown in table 3.1.

Having the matrix of pairs of *range graph* partitions, we can obtain the topological orders for each matrix entry, and then use them as inputs for the GET_SIG algorithm, obtaining a set of $\mathcal{S}^+$ for a set of pattern partitions. Before we can use the set of $\mathcal{S}^+$ to retrieve candidate windows, we must observe that, from the first partition, the

HRG_1 with VRG_1	HRG_2 with VRG_1	HRG_3 with VRG_1
HRG_1 with VRG_2	HRG_2 with VRG_2	HRG_3 with VRG_2
HRG_1 with VRG_3	HRG_2 with VRG_3	HRG_3 with VRG_3
HRG_1 with VRG_4	HRG_2 with VRG_4	HRG_3 with VRG_4

Table 3.1: Matrix of pairs of *range graph* partitions.

rest of the partitions have a displacement fixed by the dimension of the partition. With the first partition corresponding a window with ID i, a layout of width L_w and being W_w the width of the windows of extraction, the ID of a partition in the (X, Y) position in the matrix of pairs of *range graph* partitions is given by the formula:

$$ID = i + (X \times n) + (Y \times m \times (L_w – W_w + 1)) \tag{3.1}$$

Formula 3.1: ID of the window that contains a partition, with the window of ID i as reference.

Suppose a pattern partition into sub-partition of n of width and m of height. Table 3.2 shows the displacement taking as reference the window's ID at the first partition.

	X=0	**X=1**	**X=2**
Y=0	i	$i + n$	$i + (2 \times n)$
Y=1	$i + (m \times (L_w – W_w + 1))$	$i + n + (m \times (L_w – W_w + 1))$	$i + (2 \times n) + (1 \times m \times (L_w – W_w + 1))$
Y=2	$i + (2 \times m \times (L_w – W_w + 1))$	$i + n + (2 \times m \times (L_w – W_w + 1))$	$i + (2 \times n) + (2 \times m \times (L_w – W_w + 1))$
Y=3	$i + (3 \times m \times (L_w – W_w + 1))$	$i + n + (3 \times m \times (L_w – W_w + 1))$	$i + (2 \times n) + (3 \times m \times (L_w – W_w + 1))$

Table 3.2: Matrix of partitions: IDs displacement.

The table 3.2 means that, if we find a window with ID i that matches the $\mathcal{S}^+$ of the pair of HRG and VRG of the position $(0, 0)$ of matrix of *range graphs* partitions pairs, we must verify if the window with ID $i + n$ matches the $\mathcal{S}^+$ of the pair of HRG and VRG of the position $(0, 1)$ of the matrix of *range graphs* partitions pairs, and so on for the whole matrix. If one of the conditions is not satisfied, then the whole set of partitions cannot contain the pattern from where the partitions are obtained. By performing this procedure, we can split a pattern larger than the windows of $\mathcal{S}^+$ extraction and use the $\mathcal{S}^+$ of its partitions to find if a set of adjacent windows conform, as a group, the given pattern that was splitted.

Chapter 4

Experimental evaluation.

In this chapter, we describe the experiments that were executed in order to evaluate the cost of the $\mathcal{S}^+$ extraction process and the quality of the *signature* to filter our regions of a layout from. We also include an extensive evaluation of different indexing structure and their effect in the time and space costs.

4.1 Dataset.

First of all, to execute any experiment we need to have a dataset. This dataset must include a set of layouts, each of them with different characteristics in order to test the performance of the process under different data inputs. These characteristics correspond to different layout dimensions and different *hotspot* density. The same must occur with a set of *range patterns*. We need to have different patterns, each of them with different characteristics. These characteristics correspond to different possible sizes of patterns, different number of geometric shapes within the patterns, different ranges within rectangles, and different rectangles orientations.

Since each *fab* owns different IC designs that usually provide a competitive advantage in the market, it is very difficult to obtain a real design that match current technology. Due to non-disclosure agreement with Synopsys Inc.[1], we did not count with real layouts and most of the free-access examples are very old, or correspond to a reduced view of what an IC is. What it is possible to know is some of the standard characteristics that a currently used IC has. Here we can mention that a typical current IC design has area range from a few square millimeters to around 300 mm^2, while the manufacturing technology has gone under 60nm. If we take into account the regular grid representation, where each cell corresponds to the minimal printable

[1]Synopsys Inc. has partially funded this thesis work.

element (60nm), an IC design of 12mm of width and length will have a grid representation of 200.000 × 200.000 cells. Also, we can mention that transistors within an IC usually extend along the x-axis, and in less proportion, along the y-axis. This produces that the geometries in a design also typically extend along the x-axis for few nanometers, and less likely along the y-axis for less nanometers.

As the *hotspot*-detection process takes a high amount of computational resources, *fabs* usually use cluster computing for its design processes. This means that an IC design is partitioned in order to distribute the checking steps, e.g., the *hotspot*-detection process. Each one of these partition can be intended as a unique input for a node of the cluster, and it is processed in an isolated environment. In this work we will focus on these partitions. Taking this into consideration, we have built a set of layouts, whose characteristics are shown in the summary table 4.1. Each layout contains a number of *hotspots* that cover a certain area of the layout. These *hotspots* correspond to 7 *hotspots* patterns that will be presented later (see Chapter A). For each pattern, 3 different realizations were inserted into the layouts.

Layout	**Regular grid dimensions**	**Ocurrence of *hotspots***	***Hotspots* area (%)**
$A1$	4.000 × 4.000	7.997	4,998
$A2$	8.000 × 8.000	31.767	4,963
$A3$	14.000 × 14.000	97.248	4,961
$B1$	4.000 × 4.000	16.144	10,09
$B2$	8.000 × 8.000	63.647	9,944
$B3$	14.000 × 14.000	195.539	9,976
$C1$	4.000 × 4.000	24.028	15,017
$C2$	8.000 × 8.000	95.751	14,961
$C3$	14.000 × 14.000	293.389	14,968
$D1$	4.000 × 4.000	31.531	19,706
$D2$	8.000 × 8.000	127.980	19,996
$D3$	14.000 × 14.000	391.591	19,979

Table 4.1: Experimental layouts: summary table.

A similar limitation occurs with the *hotspots*. Each *fab* has its own manufacturing process for its corresponding technology. Since a competitive advantage of an IC is its reliability, *fabs* do not publicize sensible data such *hotspot* patterns. To use some *hotspot* patterns that could be considered realistic, we refer to those *hotspots* used in the literature. So, *hotspots* like those used in [4] (see figure 4.1) were converted into *range pattern* representations like the one shown in [6].

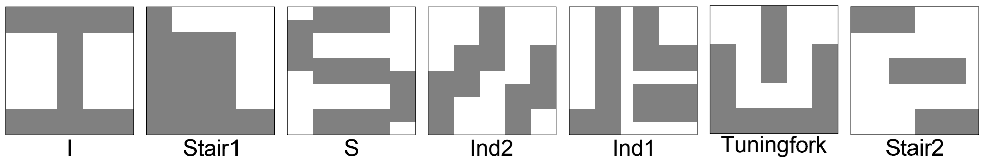


Figure 4.1: Examples of experimental *hotspots*-patterns.

Each one of these *hotspot* patterns have different characteristics that are summarized in the following table:

Pattern name	**# of rectangles**	**Orientation**	**# of $\mathcal{S}^+$**
i	3	all	1
ind1	5	all	9
ind2	5	vertical	12
s	5	all	16
stair1	3	all	1
stair2	3	horizontal	6
tuningfork	4	all	10

Table 4.2: Experimental *range patterns*: summary table.

In the following sections we run the following types of experiments:

1. We analyze different hardware sensitive strategies for *signature* extraction. The idea was to make use of the hierarchy of memory to reduce time cost.

2. We compare different indexing structure in terms of space cost and time of construction.

3. We analyze the search cost over the indexing structure.

4. We evaluate the quality of filtering portion using the *signature*, in terms of information retrieval measurements.

All of the experiments were implemented in the C++ programming language on a Linux Platform of a server with 2 processors Intel Xeon QUAD core E5620 (2,40 GHZ / 12 MB cache L3) and 64 GB RAM.

4.2 Memory hierarchy in the efficiency of *signature* extraction from layouts.

One of the key aspects that a *hotspot*-detection process must achieve is to use the computational resources efficiently. Any technique that aims to improve a *hotspot*-detection process must follow the same objective. As IC designs are big data structures, and thus its previously defined regular grid representation are also a big data structure, our designed process must be efficient in terms of time to process and memory used. The trade-off of these two aspects requires that both, data structures and data processing, should be aligned and well designed. In this work, these decisions needed to calibrate the extraction process, were taken based on an empirical comparison between different models under different circumstances.

First at all, it is important to see that each element of the $\mathcal{S}^+$ tuple, i.e. horizontal *signature* and vertical *signature*, cannot have more elements that the height or width of a window, respectively, because we cannot have more slices than rows or columns in any representation (regular and irregular representation). Each element of the horizontal(vertical) *signature* corresponds to the number of changes along a slice or a row(column), so any of these elements cannot have a value greater than $(W_w - 1)$ for the horizontal *signature* and $(W_h–1)$ for the vertical *signature*, where W_w corresponds to the window width and W_h to the window height in slices or row/columns, respectively. As windows sizes in the extraction process are typically small (from 5 to 15 regular grid cells), each element of a member of the $\mathcal{S}^+$ tuple (horizontal or vertical signature) can be codified as an INT. With this in mind, we first probe a simple and naïve extraction.

4.2.1 Layout preprocessing using a single scope.

The naïve extraction consists in calculating the irregular grid representation for each window, moving windows as was stated in 3.2. To doing so, we must read each window from the layout, what means that the layout must be in main memory. The problem is that layouts are typically expensive to store in memory due its size. One effective way to overcome this problem is to use what is defined as a scope. A scope is a portion of the layout that can be stored in main memory and allows performing all the windows extraction in the horizontal direction. Then, the scope should move the same as the window extraction process moves. Take as an example the layout of 8×8 cells used previously in our examples, and a window size of 6×6 like the one shown in figure 4.2.

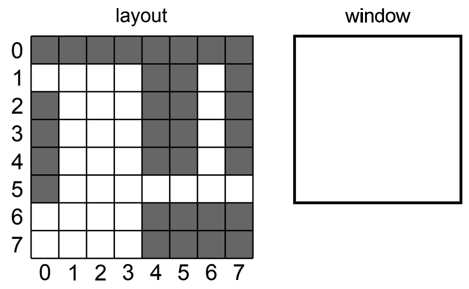


Figure 4.2: Example of a layout and a window size.

For the window extraction process we can store all the regular grid representation of the layout in main memory as a matrix, but it can overload the system memory. To avoid that, we can load first a scope like the one shown in figure 4.3 and then perform the window extraction process with the window movement along the x-axis.

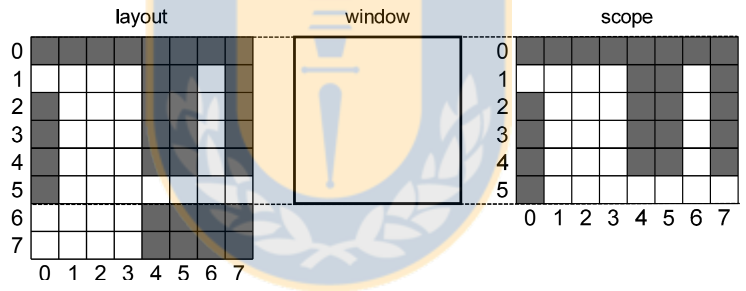


Figure 4.3: Example of a layout and its first scope.

When all the windows within the scope are extracted, we can perform a movement of the scope in the y-axis and iterate again over the x-axis, like figure 4.4 shows.

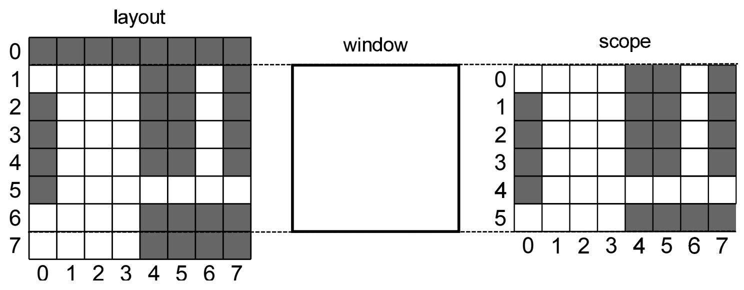


Figure 4.4: Example of a layout and its second scope.

After that, the operation repeats and we can move the scope to obtain the third and last scope (see figure 4.5). Then, the windows are extracted along the x-axis again.

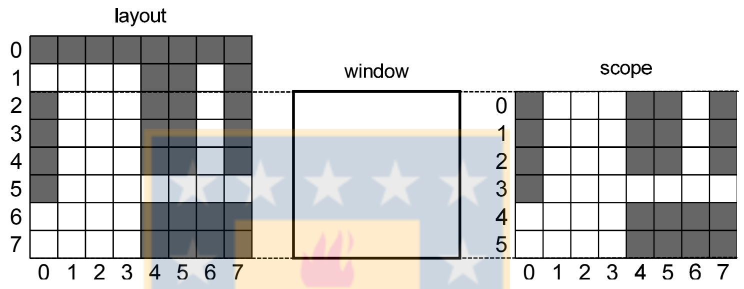
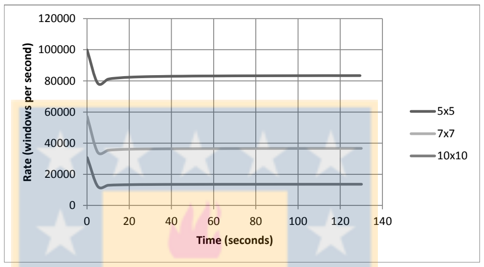


Figure 4.5: Example of a layout and its third scope.

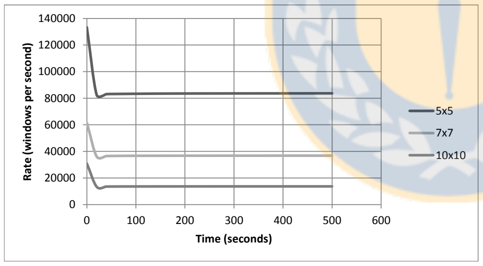
The use of a scope allows performing the window extraction process without overloading the main memory of the system. Even more, the initial time required to store the necessary data in main memory to start processing a layout is diminished. It is important to notice that it is not necessary to load an entire scope each time, excepting the first time. When the first scope moves to the second scope, we only need to remove the first row of the scope and append a new row of the layout at the end of the scope because the window extraction process moves one-cell at a time along the y-axis. The same occurs until the last scope.

By performing the previous method of window extraction, we still have not improved the execution time of the $\mathcal{S}^+$ extraction process, but we have reduced the main memory necessary to process a layout. We perform an extraction process to compare extraction time-rates between windows of 5×5, 7×7 and 10×10 over all the test layouts shown in table 4.1. As figure 4.6(a) shows, for the test layout

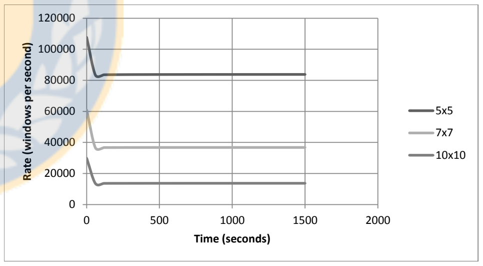
$A1(4.000 \times 4.000)$, the window extraction rate tends to be 83.500 windows per second when the size of the window is 5×5 cells. In the same figure, we can see that the window extraction rate tends to be 36.500 windows per second when the size of the window is 7×7. In the other hand, the window extraction rate tends to be 13.500 windows per second when the size of the window is 10×10. Figure 4.6(b)-(c) show us that this tendencies are not affected by the layout size since figure 4.6(b) shows the tendencies for the experimental layout $A2(8.000 \times 8.000)$ and figure 4.6(c) shows the tendencies for the experimental layout $A3(14.000 \times 14.000)$. The same tendencies are obtained by performing the window extraction process over the other test layouts. As we can see, a small change in the window dimensions can impact greatly on the rate at which the windows are extracted from the layout.



(a)



(b)



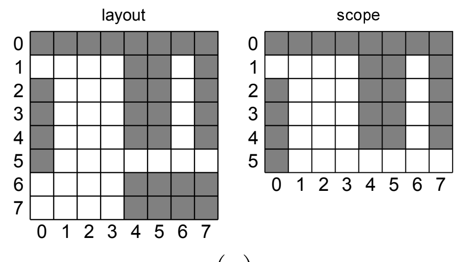
(c)

Figure 4.6: Naïve extraction rates

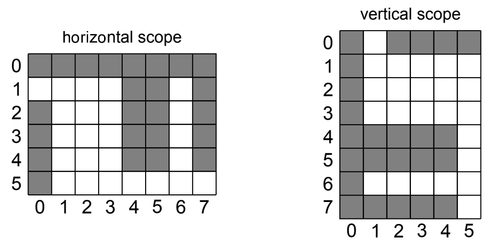
(a) - extraction rate over a 4.000×4.000 layout, (b) - extraction rate over a 8.000×8.000 layout, and (c) - extraction rate over a 14.000×14.000 layout.

4.2.2 Layout preprocessing using differentiated scopes.

Most of the matrix-oriented representations store rows (or parts of them) as contiguous blocks in memory. This allows fetching a word into the cache, which is a small but fast memory hardware. If the data requested by the processor is already in this cache, fetching the data into the processor's registers to make operations will be a fast process. On the other hand, if the data is not in the cache but in main memory, fetching the data will take a longer time making the whole process expensive, time wise. When a processor's operation requests some data which is not in any of its registers, it requests the data to the cache memory. If the data is not in the cache, it has to be loaded from the main memory. As data is not loaded individually, a fixed amount of data, i.e., a word, is fetched. Because fetching data from main memory is more expensive compared to fetching it from the cache memory, it is desirable to fetch the highest amount of useful data from main memory to the cache every time and give it the more use possible before the next data-fetching. The problem with the previously mentioned storage method, which most matrix-oriented representations use, is that when a column operation is performed, at least one fetch from main memory must occur, because two cells in different rows do not belong to the same word. This limitation set the fetching from main memory as a bottleneck in vertical matrix operations, e.g., the vertical *signature* calculation. One way to overcome this limitation is to load the scope into two different structures. One of them is a horizontally-oriented matrix and the second one is a vertically oriented matrix (see figure 4.7).



(a)

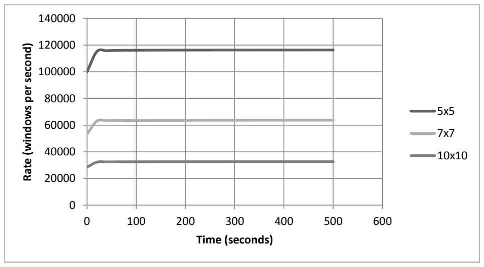


(b) (c)

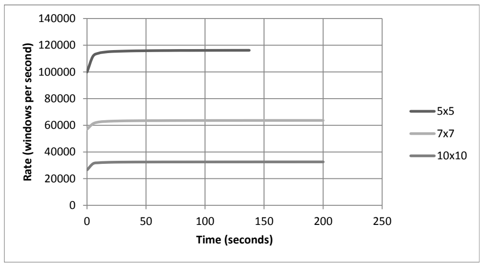
Figure 4.7: One scope and its two scopes structures.

(a) a layout with a scope, (b) its horizontal scope structure, and (c) its vertical scope structure.

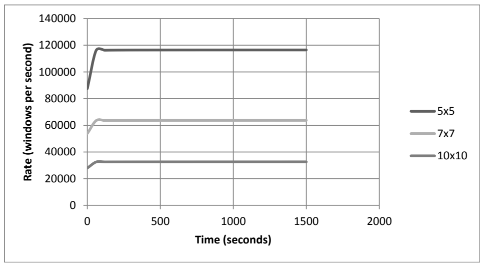
Using this two scope structures to represent the same scope, we can take advantage of the cache performance in order to reduce the time associated with the window extraction process. The same window extraction process presented before was tested using these two scope structures, meaning a slightly increased use of main memory (from $0,8MB$ for the 4.000×4.000 layout, to $1,8MB$ for the 14.000×14.000 layout), but a notable increase in the window extraction rate as figure 4.8 shows. Note that, for each scope movement along the y-axis, the vertical scope must displace a column instead of a row. The experimental results show that the costs associated with this operation are negligible compared to the improvements obtained in the whole process by using this structure.



(a)



(b)



(c)

Figure 4.8: Naïve extraction rates using two scope structures

(a) - extraction rate over a 4.000×4.000 layout, (b) - extraction rate over a 8.000×8.000 layout, and (c) - extraction rate over a 14.000×14.000 layout.

As we can see, using a simple structure that take advantage of the cache memory, we obtain an improvement that goes from around 40% when using small windows (5×5), up to 140% when using medium size windows (10×10). This difference can be explained by the way that cache memory works. Cache memory usually has three levels, i.e., $L1$, $L2$ and $L3$, being $L1$ the faster and $L3$ the slower of the three levels. When the processor requests some data, it first tries to look for it in the first level cache ($L1$). If the data is not there, it then tries to fetch it form the second level ($L2$). If the data is not there, it then tries to fetch it form the third level cache ($L3$), and if this fails, it goes up to the main memory to get the data. In other words, each failed lookup increases the time that the processor needs to fetch in the correct level of cache, or in the main memory. As a window must be processed in both orientations (i.e. horizontal and vertical) before the next window can be processed, a small window size provokes constant exchanges between horizontal and vertical data. As the cache memory is typically small compared to main memory, every time that the processor requires some data, this data competes with another data to use the cache memory. Failed lookups in cache memory caused by constant exchanges of data, and these constant exchanges itself can increase the operation of an algorithm for reasons that are not restricted to the complexity of the calculation.

4.2.3 Layout preprocessing using pivots.

Until this point, we have only optimized the process by taking advantage of the hardware and how it works. We can take into account the nature of the extraction method to obtain a better performance.

It is easy to notice that two contiguous windows share a lot of common information. In fact, if we have a window of $M \times N$, this window will share $(100 \times (N-1)/N)\%$ with its horizontally neighbor windows, and $(100 \times (M-1)/M)\%$ with its vertically neighbor windows. For both formulas, and considering M and N bigger than 1, the lowest value is 50% and tends to be 100% as the window size grows (see table 4.3).

(a)	(b)
$\frac{100(M-1)}{M}$	$\frac{100(N-1)}{N}$
$\lim_{M\to 2} \frac{100(M-1)}{M} = 50$	$\lim_{N\to 2} \frac{100(N-1)}{N} = 50$
$\lim_{M\to \infty} \frac{100(M-1)}{M} = 100$	$\lim_{N\to \infty} \frac{100(N-1)}{N} = 100$

Table 4.3: Shared area between contiguous windows.

(a) shared area formula between horizontal neighbors and its limits, (b) shared area formula between vertical neighbors and its limits.

Considering these characteristic between adjacent windows, we can reduce the time cost of the extraction if we do not calculate $\mathcal{S}^+$ for every window but only that part that is exclusive of a window and it is not shared with its previous neighbor. If we can store previous calculated $\mathcal{S}^+$ and share this information within windows, we can avoid the need to traverse an entire window for the corresponding $\mathcal{S}^+$ calculation. Here we can state some observations:

- For the horizontal displacement, as the extraction window moves one-cell to the right of the scope each time, only the first and the last vertical slices of the window could be affected. For the first vertical slice, its width may be reduced by 1, or the entire slice may disappear in the case that its width is actually 1. For the last vertical slice, its width may be incremented by 1 in the case that the new column has the same slice representation, or a new slice may be appended to its left side (remember that two adjacent slices cannot be equals by definition).

- For the vertical displacement, as the extraction window moves one-cell to the bottom of the scope each time, only the first and the last horizontal slices of the window could be affected. For the first horizontal slice, its width may be reduce by 1, or the entire slice may disappear in the case that its width is actually 1. For the last horizontal slice, its width may be incremented by 1 in the case that the new row has the same slice representation, or a new slice may be appended to its bottom side (remember that two adjacent slices cannot be equals by definition).

These statements tell us that for calculating $\mathcal{S}^+$ of a window, it is possible to only care about the peripheral slices of the previous window. If we use this idea to model the window extraction process, we can not only improve the associated time cost by reducing the data fetching but we also diminish the time cost by reducing the number of processor operations needed. As the statement suggests, we need to focus on two aspects about the peripheral slices, namely, the geometric structure of the slice, and its width in the case of vertical slices or its height in the case of horizontal slices.

Given that the horizontal displacement of the extraction window goes from the left to the right of the scope, moving one-cell a time, every first window of the scope will act as pivot. We can use a structure like the one shown in figure 4.9 to store the vertical *signature* and the vertical slices width of the first window of the scope as a tuple. In this figure, the first element of the tuple stores the changes through the vertical slices of the window as integers. The second element stores the width of each vertical slice also as integers. $VCh_1, ..., VCh_n$ correspond to the changes through the first vertical slice up to the n^{th} vertical slice, while $Vw_1, ..., Vw_n$ correspond to the width of the first vertical slice up to the n^{th} vertical slice.

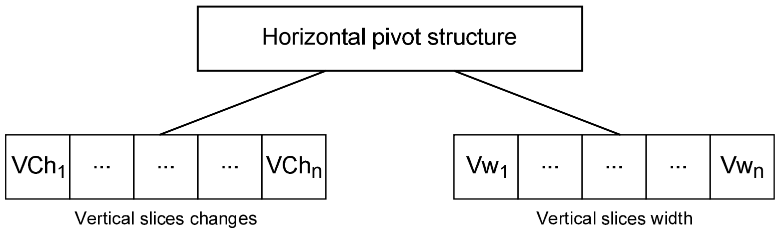


Figure 4.9: Horizontal pivot structure representation.

As the extraction window keeps moving to the right of the scope, we can obtain the vertical $\mathcal{S}^+$ of the current window of the scope (excepting the first window of the scope) with the following declarations:

1. If $Vw_1 > 1$, then Vw_1 decreases by 1 and VCh_1 remains the same. If $Vw_1 = 1$, then Vw_1 and VCh_1 are eliminated.

2. If the last column of the window is equal to the penultimate column of the window, then Vw_n increases by 1 and VCh_n remains the same. If the last column of the window is not equal to the penultimate column of the window, then a 1 is appended to the second element of the pivot, and the geometry changes along the new column is appended to the first element of the pivot structure.

3. The vertical $\mathcal{S}^+$ of the current window corresponds to the first element of the pivot structure. And the window can now be displaced to the right of the scope.

For the vertical displacement of the scope in the window extraction process we can use a similar idea but with some differences. First we have to notice that, given the nature of the scope's vertical displacement, the first window of the scope will share most of its area with the first window of the next scope (when the scope moves a position to the bottom of the layout). Given that the scope does not moves a one-cell to the bottom of the scope until all the windows of the scope were processed (all the horizontal displacements within the scope were executed), we need to "remember" the pivot of the first window of the current scope to use it with the first window of the next scope. From this point of view, we can say that in the vertical scope displacement, the pivots will work in the corresponding column of extraction windows. Due this operation model, we will need as many pivots as windows exist along the x-axis of the scope. We can represent this pivots structure as figure 4.10 shows. In this figure, there are j pivots because j windows are extracted horizontally along the layout's width. Each vertical pivot has a similar structure to the horizontal pivot structure. Here HCh_i, with $1 \leq i \leq n$, corresponds to the changes through the i^{th} horizontal slice, while Hh_i, with $1 \leq i \leq n$, corresponds to the height of the i^{th} horizontal slice of the window represented by the corresponding pivot in the vertical pivot structure.

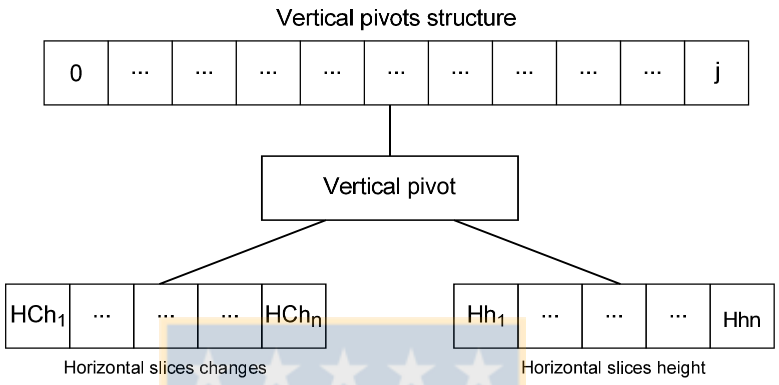


Figure 4.10: Vertical pivots structure representation.

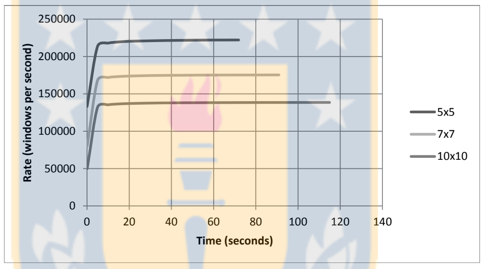
Given that each window within a layout is identified by a correlative number starting by 0, we can identify the corresponding window's pivot in the vertical pivot structure by using this window identificator. A window identified by the Wi identificator will use the $(Wi \text{ MOD } (Layouts'\ width - Windows'\ width + 1))$ pivot of the vertical pivot structure. As the extraction scope keeps moving to the bottom of the layout, we can obtain the horizontal $\mathcal{S}^+$ of the current window of the scope (excepting for those in the first scope of the layout) with the following declarations:

1. If $Hh_1 > 1$, then Hh_1 decreases by 1 and HCh_1 remains the same. If $Hh_1 = 1$, then Hh_1 and HCh_1 are eliminated.

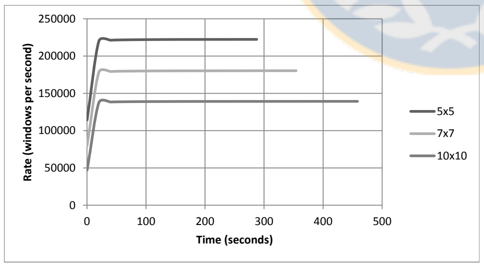
2. If the last row of the window is equal to the penultimate row of the window, then Hh_n increases by 1 and HCh_n remains the same. If the last row of the window is not equal to the penultimate row of the window, then a 1 is appended to the second element of the pivot and the corresponding changes value through the row is appended to the first element of the pivot structure.

3. The horizontal $\mathcal{S}^+$ of the current window corresponds to the first element of the pivot.

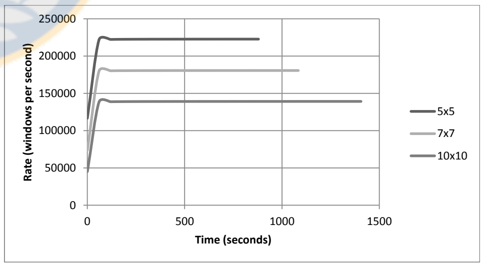
Using the horizontal and vertical pivots to calculate $\mathcal{S}^+$ of a window within a scope, and representing this scope using two structures (i.e., horizontal scope structure and vertical scope structure), we are not only taking advantage of the cache performance but also of the extraction process nature. For these reasons, the time costs are dramatically reduced as figure 4.11 shows. In the figure 4.11(a) we can see that the windows extraction rate tends to be 222.100 windows per second when the size of the window is 5×5 cells. In the same figure, we can see that the windows extraction rate tends to be 175.300 windows per second when the size of the window is 7×7, while the windows extraction rate tends to be 138.500 windows per second when the size of the window is 10×10. The same tendencies are obtained by performing the windows extraction process over the other test layouts.



(a)



(b)



(c)

Figure 4.11: Smart extraction rates using two scope structures and pivots for $\mathcal{S}^+$ calculation.

(a) - extraction rate over a 4.000×4.000 layout, (b) - extraction rate over a 8.000×8.000 layout, and (c) - extraction rate over a 14.000×14.000 layout.

From what follows in this document, this last extraction method will be used in the next experiments.

4.3 Indexing structures.

The previous section shows different $\mathcal{S}^+$ extraction methods and how we can improve its cost time associated. However, any extraction process is worthless if we do nothing with the extracted data. We can use it immediately or we can store it to use it later. This work aims to contribute with the *hotspot*-detection process, not to be a *hotspot*-detection process by itself. From this perspective, all the useful data obtained from the $\mathcal{S}^+$ extraction process must be stored to be used lately in any process that can give it a use. In our case, this later process corresponds to a layout's area pre-filtering by using windows and its $\mathcal{S}^+$. To achieve this objective, a suitable storage structure must be employed. This storage structure must meet the following goals:

- Must group, at least, all those windows with the same $\mathcal{S}^+$.
- Must allow a direct access to any group based in the $\mathcal{S}^+$.
- Should use the less possible memory space.

In addition to the previous goals we can add some desirable characteristics:

- Each window group must be as homogeneous as possible, i.e., ideally should be only composed by windows with the same $\mathcal{S}^+$.
- The access to any group must be computational cheap, time and memory wise.
- Should use the less possible memory space without compromising time costs to access and retrieve data.
- The structure creation and population cost should be the smallest possible, time wise.

As a layout can contain several different windows on it (($layout\_width$ - $window\_width$ + 1)*($layout\_height$ - $window\_height$ + 1) to be exact), and each window can be mapped to a specific $\mathcal{S}^+$, we can have several different $\mathcal{S}^+$ values within a layout. If we want to group windows based on the corresponding $\mathcal{S}^+$ mapping, it is natural to think in an indexed structure. In our case, there are a variety of data structures that can fit the requirements of an indexed structure, but we can mention the most used and accepted: Search trees and Hash tables. With both having advantages and disadvantages, we tested the performance of various different implementations, measuring time cost of creating and populating the structures, as well as the memory usage in the process. Similar to the previous section, here we will show the extraction time rates for each implementation, but now including the insertion of windows in the structure. We will measure the memory usage as the structure is populated with the objective of showing tendencies in both measurements.

4.3.1 Using a binary search tree.

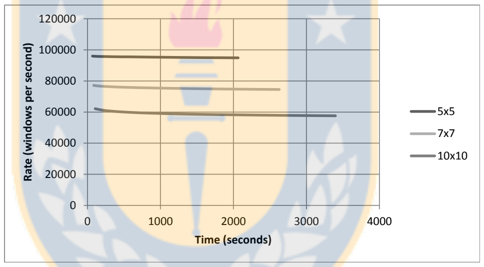
The first implementation of our work is a self-balancing binary search tree. This implementation was built using MAP[2] of the STL library[3]. Probably most common self-balancing trees are red-black trees and AVL trees. For both cases, the insert/remove operation (insert is the most intensive operation in our work) are $O(log\ n)$ being n the number of nodes in the tree. Inserting/removing in a red-black tree may violate the properties of a red-black tree. Restoring their properties requires a small number of $O(log\ n)$ or amortized $O(1)$ of color changes and no more than three tree rotations. Being an AVL tree a more rigidly balanced tree than red-black trees, the retrieval operation is faster but the insertion and removal operations are slower than in a red-black tree. For this reason, the MAP structure was implemented using a red-black tree. In this red-black tree, each node of the tree represents one value of $\mathcal{S}^+$, and points to a $VECTOR$[4] of INT which stores the IDs of the windows that were mapped to the $\mathcal{S}^+$ represented by the corresponding node. Each entry corresponds to the concatenation of the two elements in the corresponding $\mathcal{S}^+$ of a window, i.e., the horizontal and vertical *signature* over the irregular grid representation of the window.

It is important to notice that for the red-black tree creation, strings must be

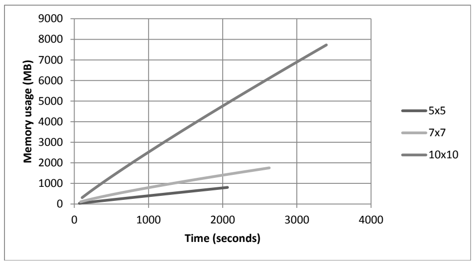
[2]http://www.cplusplus.com/reference/map/map/
[3]STL stands for Standard Template Library.
[4]http://www.cplusplus.com/reference/vector/vector/

compared by the < (less-than) operator. The less-than operator does a lexicographical comparison on the strings. This compares strings in the same way that they would be listed in dictionary order. This means that the less-than operator iterates character-by-character on the strings, comparing its corresponding ASCII values.

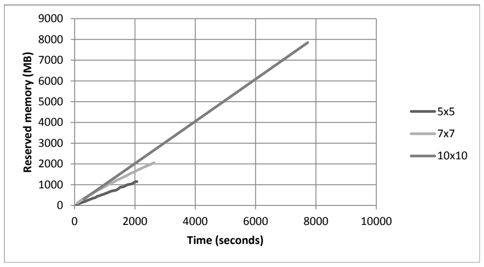
Figure 4.12(a) shows the extraction rates while the MAP structure is populated. It also shows the memory used by the structure through time (see figure 4.12(b)-(c)). Here, we can see two different memory measurements: used memory and reserved memory. The used memory means how much memory is being effectively used by the structure, while the reserved memory means how much memory is reserved for the structure. This difference is caused by how some structures are implemented in the STL library. Many data structures (e.g. $VECTORS$) receive some memory space when they are declared. When the data within the structure fills all the reserved space, new space must be incorporated. This step usually doubles the previous reserved memory space, so we can have structures with a certain reserved memory space, even when the user data uses less memory space.



(a)



(b)



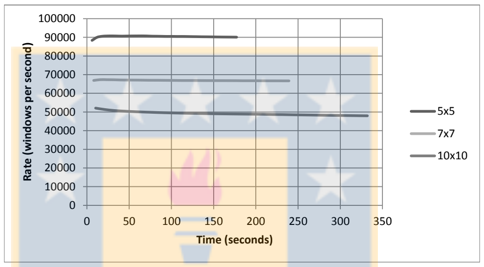
(c)

Figure 4.12: MAP structure: Extraction and population, memory usage and reserved memory over time.

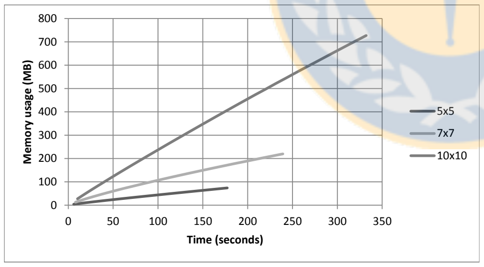
(a) extraction and population rate, (b) memory usage, and (c) reserved memory over a 14.000 × 14.000 layout.

As we can see in figure 4.12(a), as far as window size increases, the extraction rate decreases. For the 14.000×14.000 layout shown in the figure, when the extraction window size has a size of 5×5, the extraction-with-population rate tends to be 94.800 (100%) windows per second, but when the size of the window of extraction increases to 7×7, the extraction-with-population rate tends to be 74.500 (78.58%) windows per second. Even more, when the size of the window of extraction increases to 10×10, the extraction-with-population rate tends to be 57.600 (60.76%) windows per second. The downfall of the extraction rate is not only affected by the $\mathcal{S}^+$ calculation as we saw in the previous chapter (larger windows take more time to extract its $\mathcal{S}^+$), but also by the comparison needed to find the location of a $\mathcal{S}^+$-string within the MAP structure. As larger windows tend to have longer $\mathcal{S}^+$-string representation due to the increase in the number of slices, they usually require more time to compare its $\mathcal{S}^+$-string representation. Figure 4.12(b) shows that a little increase of the window size (from 5×5 to 7×7 and to 10×10) greatly impacts the memory used by the MAP structure. As the size of the window of extraction increases, the number of different values of $\mathcal{S}^+$ increases (it is less likely having two different configurations with the same value of $\mathcal{S}^+$). This means that we will have more tree nodes in the MAP structure, but less window identifiers in each node. Given that the identifiers are only integers, but the tree nodes are a structure containing the value of $\mathcal{S}^+$, and a set of pointers to the VECTOR structure containing the windows identifiers, each tree node uses more memory than a window identifier (4bytes vs 64bytes in our implementation). In conclusion, more tree nodes with less windows identifiers mean anyways more memory usage by the whole structure. The same occurs in the measurement of the reserved memory, but the tendencies with different window's size are more similar. If we compare the gradients of the tendencies between figure 4.12(b) and figure 4.12(c), we can say that larger windows's sizes mean more reserved memory for the MAP structure, but the impact in the trend of the increased memory use is less.

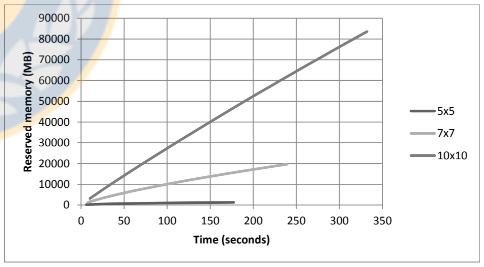
It was previously mentioned, when the reserved memory of a structure is totally filled, new memory need to be assigned. As MAP and VECTOR structures in STL implementation use contiguous memory blocks, when new memory is assigned to a structure, it probably means a memory reallocation. Reallocating memory is a costly task that should be avoided. From now, we will refer to the amount of elements within a structure as the *size* of the structure, and the reserved memory space for elements as the *capacity* of the structure. In our implementation, the default initial size is zero (when a structure is declared), and the initial capacity is one. Inserting elements increases size by one, but capacity is only affected when the size of the structure reaches the structure capacity. In that case, the structure doubles its capacity and if the contiguous memory cannot allocate the new assigned memory space, a reallocation occurs. We can try to avoid unnecessary initial reallocations by altering the default initial capacity to a larger one. If we set a vector's default initial capacity equals to $MAX(layout\_width, layout\_height)/2$ we obtain the following statistics shown in figure 4.13



(a)



(b)



(c)

Figure 4.13: MAP structure with initial reserved memory: Extraction and population, memory usage and reserved memory over time.

(a) extraction and population rate, (b) memory usage, and (c) reserved memory over a 4.000×4.000 layout.

Figure 4.13 shows the extraction rate and memory usage over a 4.000×4.000 layout. Figure 4.13(c) shows that, starting to populate the MAP structure with an initial reserved memory space will lead us to a high amount of reserved memory (up to 80GB) after a few minutes of processing. This enormous use of reserved memory impedes us to process a 14.000 $\times$ 14.000 layout. Even more, if we compare figure 4.12(a) with figure 4.13(a) we can notice that our assumption was wrong. Start populating the MAP structure with a larger amount of reserved memory not only uses an impractical amount of memory but also diminishes the extraction rate. In our case, this downgrade of the extraction speed was determined by the memory swapping needed to manage such big amount of memory space.

4.3.2 Using two binary search trees.

Our second implementation was a dual-MAP structure. This dual-MAP structure aims to use the horizontal $\mathcal{S}^+$ and vertical $\mathcal{S}^+$ as search key, but separately. The basic concepts are similar to the previous implementation: each MAP structure is implemented as a red-black tree, and the entries are strings representations of the corresponding $\mathcal{S}^+$ element (horizontal $\mathcal{S}^+$ or vertical $\mathcal{S}^+$) of a window. In this implementation, each tree node of the first MAP structure (let's call it the outter MAP) groups all the windows with the same horizontal $\mathcal{S}^+$. A tree node of the first MAP structure points to a second MAP structure (let's call it the inner MAP), that groups all the windows with the same vertical $\mathcal{S}^+$. In other words, the outter MAP groups windows with the same horizontal $\mathcal{S}^+$, while the inner MAP groups windows with the same vertical $\mathcal{S}^+$, and all of these groups share the same horizontal $\mathcal{S}^+$. Similar to the previous implementation, each node of the inner MAP structure corresponds to a VECTOR of windows ids. Figure 4.14 shows a visualization example of this dual-MAP structure.

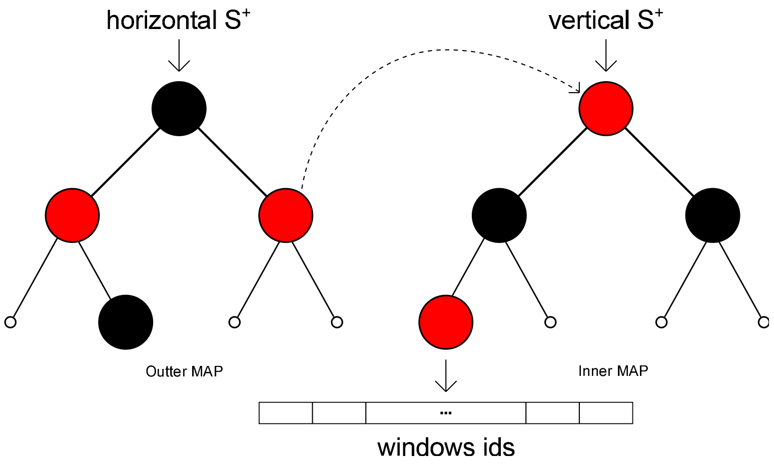


Figure 4.14: Vertical pivots structure representation.

By using this structure to extract and store windows over a 14.000×14.000 layout, we obtain the extraction rates, memory usage and reserved memory over time shown in figure 4.15

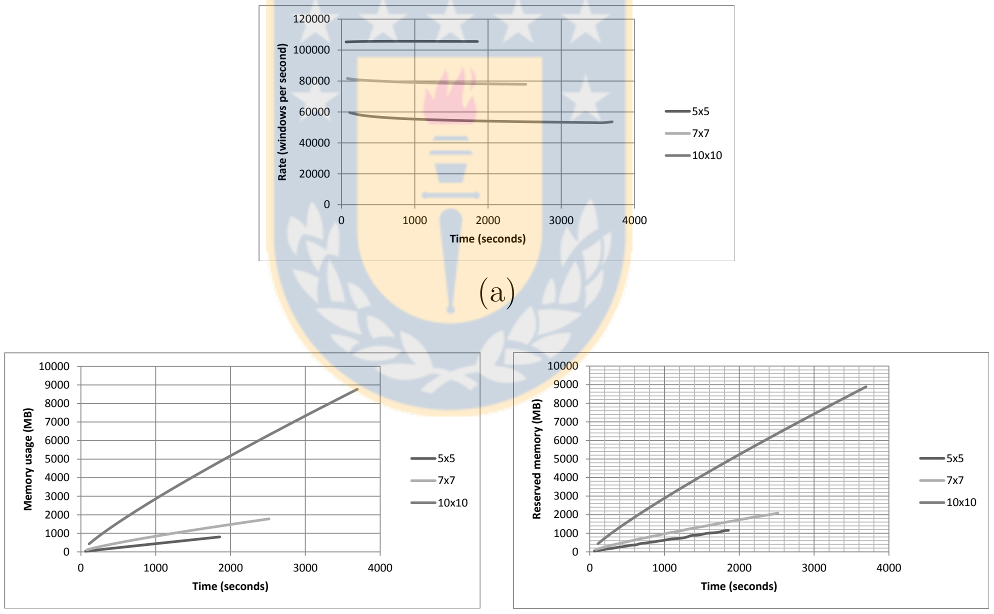


Figure 4.15: Dual-MAP structure: Extraction and population, memory usage and reserved memory over time.

(a) extraction and population rate, (b) memory usage, and (c) reserved memory over a 14.000×14.000 layout.

If we compare figure 4.15(a) with figure 4.12(a) we can conclude that using a dual-MAP structure improves the extraction-and-population rate by around a 11% when the size of the window of extraction is of 5×5 (from 95.000 windows per second up to 105.000 windows per second approximately), around a 4% when the size of the window of extraction is of 7×7 (from 75.000 windows per second up to 77.900 windows per second), and around a 1% when the size of the window of extraction is of 10×10 (from 58.000 windows per second up to 58.600 windows per second). By comparing figure 4.15(b)-(c) with figure 4.12(b)-(c), respectively, we can notice that using a dual-MAP structure does differ much in space from using a simple-MAP structure. The incremental tendencies keeps very similar, and the memory use remains almost the same, except when the window size is of 10×10, when the total memory use and reserved memory increments in around 1GB of space. The extraction rate speed-up is explained by the fact that each MAP structure in the second implementation (the outter MAP for the horizontal $\mathcal{S}^+$ and the inner MAP for the vertical $\mathcal{S}^+$) has, individually, less tree nodes stored compared with the MAP structure of the first implementation. This occurs because the grouping capacity is greater than the first implementation, because the $\mathcal{S}^+$ is separated between its two elements (horizontal and vertical). Less nodes in each MAP structure means faster insertion times. From the point of view of memory vs extraction time rates, if we try the same initial memory reserve approach, we obtain similar results that in the previous implementation: a small decrease in the extraction rate and a big increase in the memory usage.

4.3.3 Using has functions.

The two previous implementations use binary search trees as $\mathcal{S}^+$ storage structure. As we have said, hash tables are also a widely used indexing structure and it is desirable to compare how these two kinds of structures behave in the window extraction process. Hash tables are associative structures that link a key with one or more values. This work transforming the key into a hash. A hash is a value that identifies the position (bucket) where the hash table locates the desired value. Hash tables store information in pseudo-random positions, so the orderly access to its content is quite slow. Compared with other data structures such as self-balanced binary trees, hash tables have higher average search time, but the information is not sorted. As we can expect, hash tables depend highly on the selected hash function that maps keys into hash values. If this hash function is not well designed, some buckets of the hash table can be extremely saturated of entries due to hash collisions, while other buckets can be very unused. This saturation of some buckets of the hash table is known as value crowding, and its occurrence must be avoided. In our work, we tested 3 different hash functions, each one over 3 different table sizes.

First hash function.

The first hash function (hash1) was a very simple hash, whose pseudo-code is presented in algorithm 5. In this algorithm, the horizontal and vertical $\mathcal{S}^+$ lists start at the 1^{th} position and the *.position* property returns the position of a list element.

Algorithm 5 Hash-1 algorithm.

```
   Input The horizontal signature and the vertical signature as lists, and the hash table size
   Output The hash as an integer
1: $horizontal\_hash \leftarrow 0$
2: $vertical\_hash \leftarrow 0$
3: for each value $v$ in the horizontal signature do
4:     $horizontal\_hash \leftarrow horizontal\_hash + (v * v.position)$
5: for each value $v$ in the vertical signature do
6:     $vertical\_hash \leftarrow vertical\_hash + (v * v.position)$
7: return $(horizontal\_hash \times vertical\_hash)$ MOD $hash\_table\_size$
```

Second hash function.

The second hash function (hash2) was a little more complex hash that uses two prime numbers to distribute the hashes. Its pseudo-code is presented in the algorithm 6. In this algorithm, the MAX function returns the maximum value between two parameters (e.g., $MAX(2,5)$ will return 5) and the *.length* property returns the number of elements in a list. Similar to the first hash, the horizontal and vertical $\mathcal{S}^+$ lists start at the 1^{th} position and the *.position* property returns the position of a list element.

Algorithm 6 Hash-1 algorithm.

```
    Input The horizontal signature and the vertical signature as lists, and the hash table size
    Output The hash as an integer
1: $hash \leftarrow 17$
2: $iterator \leftarrow MAX(horizontal\_signature.length, vertical\_signature.length)$
3: for $i$=1..$iterator$ do
4:     if there exists horizontal_signature [$i$] then
5:         $hash \leftarrow (hash \times 31) + horizontal\_signature$[i]
6:     if there exists vertical_signature [$i$] then
7:         $hash \leftarrow (hash \times 31) + vertical\_signature$[i]
8: return $hash$ MOD $hash\_table\_size$
```

Third hash function.

The third and last hash function (hash3) is presented in the algorithm 7 and it was a modification of the second hash function that includes a percentage of bucket shifting. Similar to the second hash function, the MAX function returns the maximum value between two parameters (e.g., $MAX(2, 5)$ will return 5) and the *.length* property returns the number of elements in a list. The horizontal and vertical $\mathcal{S}^+$ lists start at the 1^{th} position and the *.position* property returns the position of a list element.

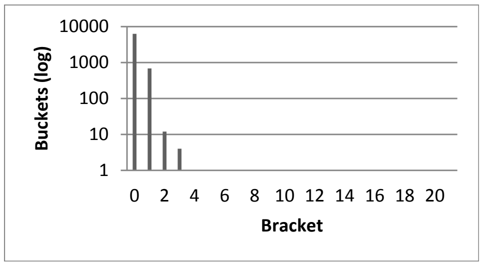
Algorithm 7 Hash-1 algorithm.

```
    Input The horizontal signature and the vertical signature as lists, and the hash table size
    Output The hash as an integer
1: $hash \leftarrow 17$
2: $iterator \leftarrow MAX(horizontal\_signature.length, vertical\_signature.length)$
3: for $i$=1..$iterator$ do
4:     if there exists horizontal_signature [$i$] then
5:         $hash \leftarrow (hash \times 31) + horizontal\_signature$[i]
6:     if there exists vertical_signature [$i$] then
7:         $hash \leftarrow (hash \times 31) + vertical\_signature$[i]
8: return $(hash+hash/2)$ MOD $hash\_table\_size$
```

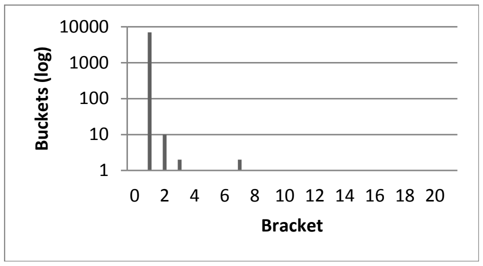
Given these three hash functions, we compare its behavior taking into account three different variables: the layout (different layout sizes and different layout geometries), the size of the windows of extraction, and the size of the hash table. We present the different results over a 14.000 × 14.000 layout, because the three hash functions behaves very similar over the others layouts of the dataset.

Comparing hash functions.

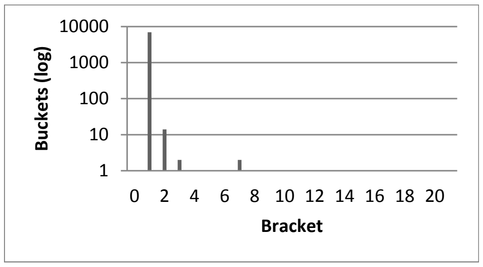
The first experiment was configured to use windows of extraction of 5×5, being the total number of windows of 195.888.016, and the size of the hash table was set to 50% of the layout's dimension (7.000 in this case). We first compare the windows-per-bucket distribution on the hash table by using the three different hash functions. A good hash function must avoid having buckets with very low windows stored, or buckets with a high number of windows. The minimum windows per bucket using the first hash function were 0 (0%). It means that there exist buckets that do not contain any window. For the second and third hash functions, the minimum windows per bucket were 105 ($5.3e^{-5}$%) and 71 ($3.6e^{-5}$%), respectively. The maximum windows per bucket using the first hash function were 39.221.398 (20%). That means that exist at least one bucket that contains 20% of the total of windows extracted from the layout. The maximum windows per bucket using the second and the third were 11.899.996 (6%) and 11.902.009 (6%), respectively. To present a histogram of windows per bucket, we set brackets with ranges of 1.961.069 windows per bucket (0-to-0, 1-to-1.961.070, 1.961.071-to-3.922.140, and so on). By using these ranges, we obtain the following histograms.



(a)



(b)



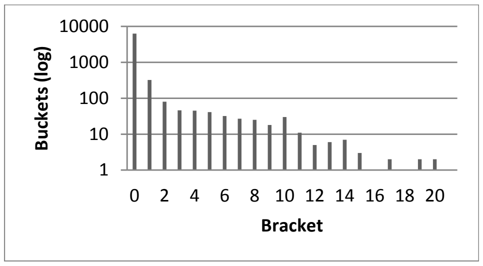
(c)

Figure 4.16: Windows extraction over a 14.000 $times$14.000 layout: Windows per bucket histogram using a hash table of 7.000 buckets.

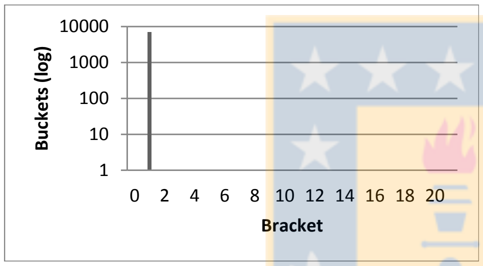
(a) extraction histogram using hash1, (b) extraction histogram using hash2, and (c) extraction histogram using hash3.

As we can see in figure 4.16(a), by using hash1 to populate the hash table, most of the buckets are cataloged in the 0^{th} bracket. That means that most of the buckets have zero windows. Comparing figure 4.16(b) and figure 4.16(c), it is possible to say that hash2 and hash3 behave very similar, cataloging most of the buckets in the 1^{st} bracket. That means that both hash functions distribute windows along the hash table, making most of the buckets to store from 1 to 1.961.070 windows. For all three hash functions, the total calculation time (sum of computation time of all windows) was of 98,7 seconds for hash1 function, 103,17 seconds for the hash2 function and 103,36 seconds for the hash3 function.

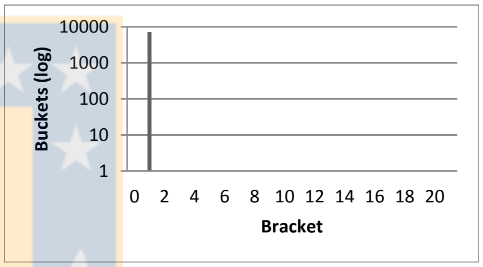
The second comparison was the number of different $\mathcal{S}^+$ within buckets. A good hash function must avoid hash collisions. That means that a good hash function must have only few different $\mathcal{S}^+$ within a bucket, being a perfect hash the one that can have one $\mathcal{S}^+$ per bucket. The minimum $\mathcal{S}^+$ per bucket using the first hash function where 0, while the maximum $\mathcal{S}^+$ per bucket using the same hash function were 3.293. That means that using the hash1 function there exist buckets with no window (and therefore no $\mathcal{S}^+$), but also there exist buckets with a great number of collisions (3.293 different $\mathcal{S}^+$ are mapped to the same bucket). By using the hash2 function, we have a minimum per bucket of 13 and a maximum $\mathcal{S}^+$ per bucket of 93. On the other hand, using the hash3 function we have a minimum $\mathcal{S}^+$ per bucket of 17 and a maximum $\mathcal{S}^+$ per bucket of 89. Similar to previous histograms, here we set brackets of ranges of 164 $\mathcal{S}^+$ per bucket, obtaining the following histograms.



(a)



(b)

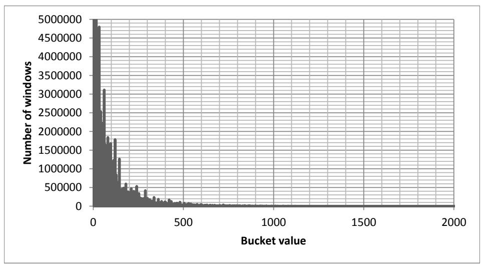


(c)

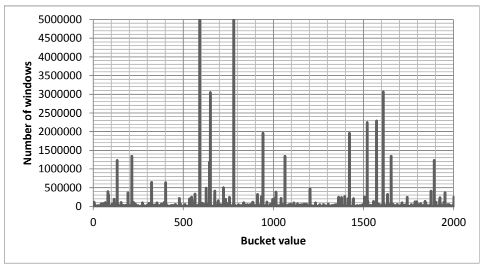
Figure 4.17: Windows extraction over a 14.000 $\times$ 14.000 layout: Different $\mathcal{S}^+$ per bucket histogram using a hash table of 7.000 buckets.

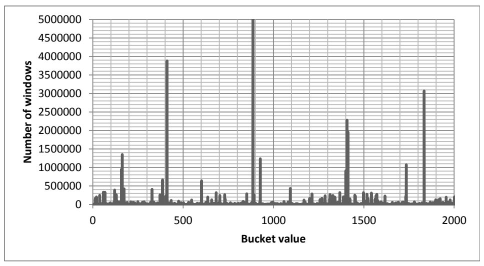
(a) extraction histogram using hash1, (b) extraction histogram using hash2, and (c) extraction histogram using hash3.

As we can see in figure 4.17(a), by using the hash1 function we have many buckets containing zero $\mathcal{S}^+$, while using hash2 and hash3 (figure 4.17(b) and figure 4.17(c) respectively) we have most of the buckets containing from 1 to 165 different $\mathcal{S}^+$. At this point, we can say that hash2 and hash3 behave very similar. The following figure (see figure 4.18) shows a zoom of the histogram of the relation windows-per-bucket without the bracket classification.



(a)





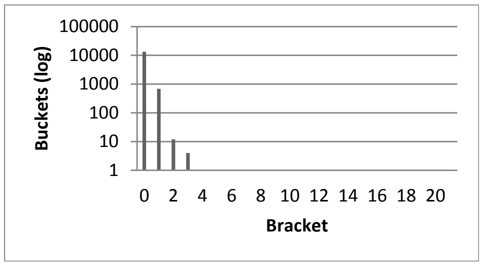
(b) (c)

Figure 4.18: Zoom of the histogram of windows per bucket for the three different hash functions.

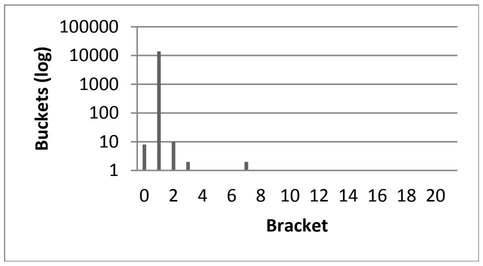
(a) zoom of the histogram using hash1, (b) zoom of the histogram using hash2, and (c) zoom of the histogram using hash3.

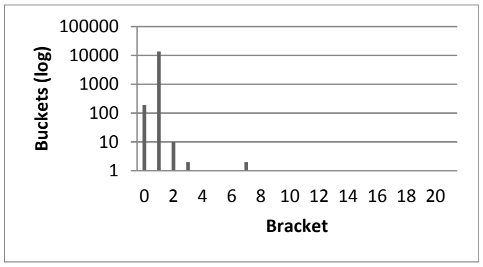
As we can see, the distribution of the hash3 is more homogeneous than the distribution of the hash2. The hash1 function distributes windows crowding buckets of low values.

The second experiment was configured to use the same size of windows of extraction (5×5), but the size of the hash table was set to 100% of the layout's dimension (14.000 in this case), while the third experiment was configured to use a table size of 150% of the layout's dimension (21.000 buckets in this case). The idea is to compare how the three different hash functions behave with a bigger hash table. Results are quite similar to the previous experiment, where hash1 has the worst distribution, hash2 a normal distribution and the hash3 function has the better and more homogeneous distribution. Figure 4.19 shows the windows per bucket using a bracket categorization of 1.961.069 and figure 4.20 shows the $\mathcal{S}^+$ per bucket using a bracket categorization of 165 for the hash table of 14.000 buckets. For all three hash functions, the total calculation time was of 99,27 seconds for hash1 function, 103,02 seconds for the hash2 function and 102,72 seconds for the hash3 function.



(a)

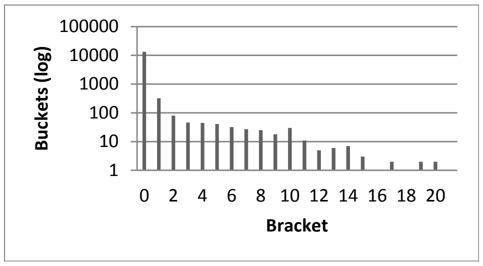




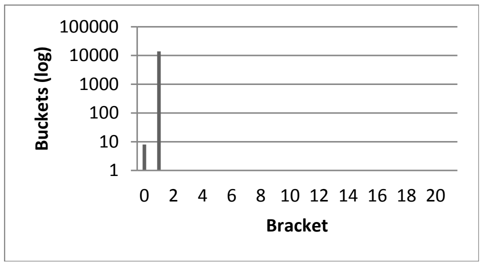
(b) (c)

Figure 4.19: Windows extraction over a 14.000 *times*14.000 layout: Windows per bucket histogram using a hash table of 14.000 buckets.

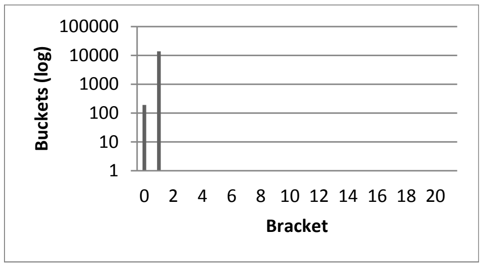
(a) extraction histogram using hash1, (b) extraction histogram using hash2, and (c) extraction histogram using hash3.



(a)



(b)



(c)

Figure 4.20: Windows extraction over a 14.000 $times$14.000 layout: Different $\mathcal{S}^+$ per bucket histogram using a hash table of 14.000 buckets.

(a) extraction histogram using hash1, (b) extraction histogram using hash2, and (c) extraction histogram using hash3.

Finally, the hash functions behavior remains the same when the buckets of the hash table are incremented to 150% of the layout's dimension (21.000 buckets in this case). Even more, this behavior remains when the size of the window of extraction are incremented to 7×7 and to 10×10. From all of the previous empirical results, the hash3 function was used as the hash function in the hash table implementations that will be presented below, because has a noticeable better distribution with a slightly higher time cost of calculation.

4.3.4 Space cost using hash table.

Our first hash table implementation aims to deliver a simply but very fast structure for window extraction. As we do not know all the possible uses of the windows extraction that a user can achieve but only we are focusing on our own windows pre-filtering process, it would be desirable to have a simply but fast structure. For this case, we trust in the quality of the hash function. As we have seen in the previous experiments, the hash3 function behaves well, with a more homogeneous distribution than hash1 and hash2 functions. However, it is far from being a perfect hash. This implementation consists in a hash structure like the shown in figure 4.21.

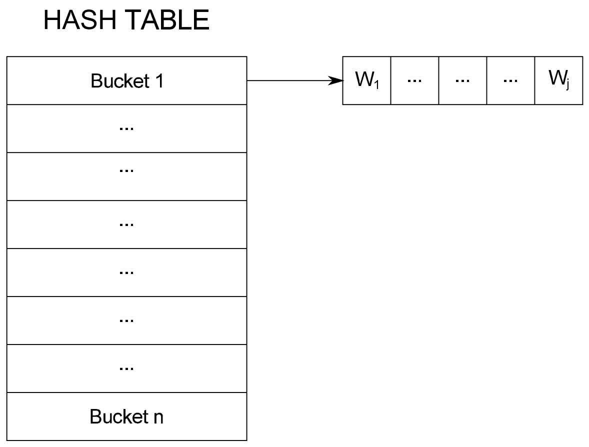
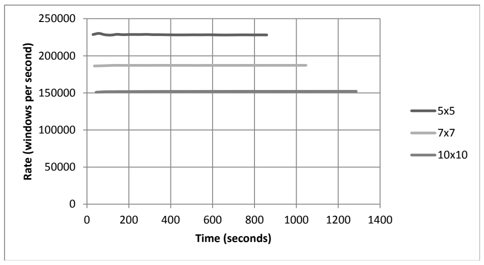


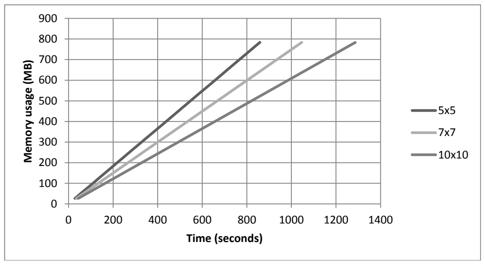
Figure 4.21: Simply hash structure representation.

Every bucket points to a vector, where windows's ids are stored.

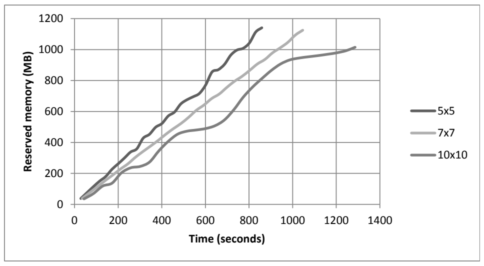
By using the hash3 as hash function, we obtain extraction time rates, memory usage and reserved memory over time as figure 4.22 shows.



(a)



(b)



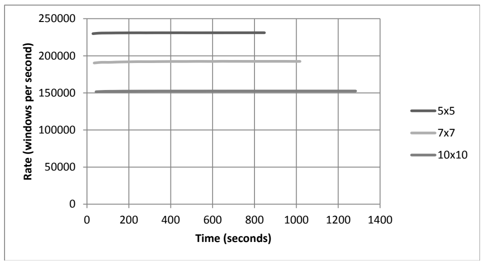
(c)

Figure 4.22: Simply hash structure: Extraction and population, memory usage and reserved memory over time.

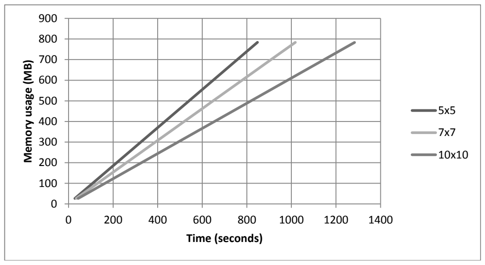
(a) windows extraction rate, (b) memory usage over time, and (c) reserved memory over time.

As we can see in figure 4.22(a), using a simply hash structure we have a really fast extraction method. Given that we have very similar extraction rates using this structure and the extraction rates without populating any structure (see figure 4.11(a)), we can say that the insertion cost is negligible compared with the $\mathcal{S}^+$ calculation cost. If we focus on the memory usage, we have a structure with a very low memory cost that not exceeds the amount of 1,3GB of reserved memory.

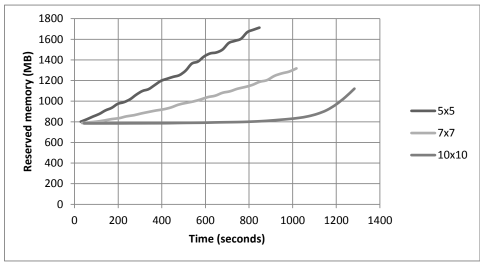
Given the results of extraction rates and memory usage, it may not be necessary to increase the initial reserved memory, but as our work is based on empirical results, it is necessary to know the behavior of the structure under some different settings. With this in mind, the same experiment was configured to start each bucket of the hash table as a VECTOR with initial reserved memory of 100% of the layout dimension (14.000 elements in this case). We obtain the extraction rates, memory usage and reserved memory over time as the shown in figure 4.23.



(a)



(b)



(c)

Figure 4.23: Simply hash structure with initial reserved memory: Extraction and population, memory usage and reserved memory over time.

(a) windows extraction rate, (b) memory usage over time, and (c) reserved memory over time.

As we can see in figure 4.23(a), extraction rates are practically not affected by incrementing the initial reserved memory of the table buckets. Figure 4.23(b) shows us that, as expected, the user data in the structure remains the same, while figure 4.23(c) shows us that the total reserved memory increases by around 50% when the size of the window of extraction is of 5×5 (from 1,1GB up to 1,7GB), 17% when the size of the window of extraction is of 7×7 (from 1,1GB up to 1,3GB), and 10,5% when the size of the window of extraction is of 10×10 (from 1GB up to 1,1GB). In conclusion, similar to previous experiments, incrementing the initial reserved memory does not provides any advantage in the extraction process. One of the problems with this simply hash structure is that we rely entirely on the hash distribution. There is no way to distinguish two windows that are stored in the same bucket but have different $\mathcal{S}^+$ (e.g., if a $\mathcal{S}^+$ collision exists), because the only way to identify a window is its hash value. For this reason, a combined structure was implemented using hash table and binary search trees.

4.3.5 Space cost using hash table plus binary search tree.

This structure aims to store windows in a set of binary search trees, to which windows are addressed using a hash function. It is implemented by a hash table, where each bucket of the table points to a red-black tree (MAP structure). Each node of these MAPs points to a VECTOR where windows's IDs are stored. The hash map address a $\mathcal{S}^+$-string representation to a given MAP, and that MAP separates all the different $\mathcal{S}^+$ that collide under the hash3 function. Each node of these MAPs structure stores all the windows's IDs of the windows with the same $\mathcal{S}^+$ in a VECTOR structure. Figure 4.24 shows a representation of this implementation.

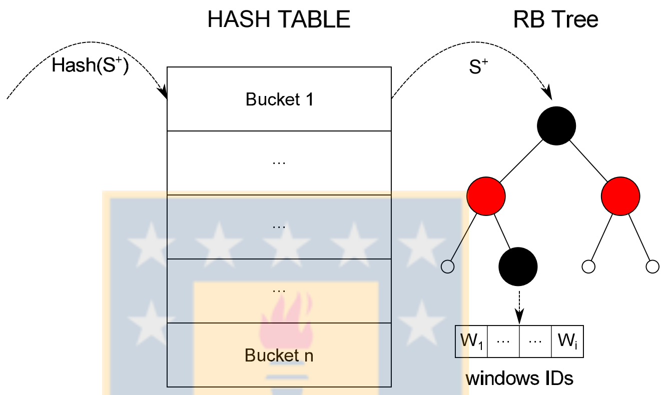
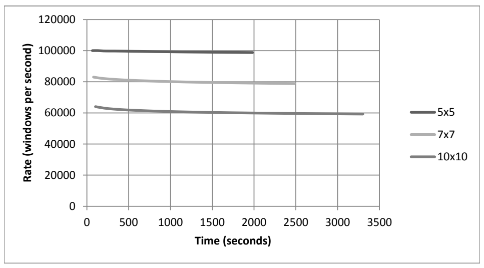
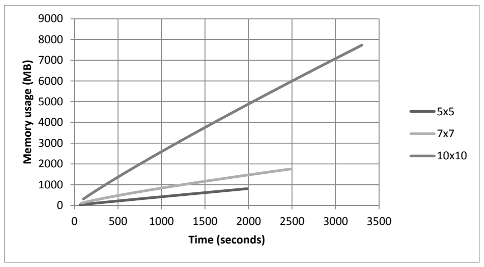


Figure 4.24: Representation of a structure using a hash table and a binary search tree.

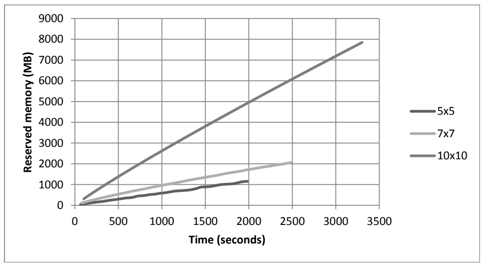
The idea is to combine the good grouping of the MAP structure, but reducing the number of the nodes of the trees (hence, reducing the insertion and querying times), by addressing only few $\mathcal{S}^+$ to the tree instead of all of the $\mathcal{S}^+$ in the layout. The hash table was configured to 100% of the layout's dimension (14.000 buckets in this case) and no initial reserved memory was used for the VECTORS. By using this structure, we obtain the extraction rates shown in figure 4.25(a). Memory usage and reserved memory over time are shown in figure 4.25(b) and (c), respectively.



(a)



(b)



(c)

Figure 4.25: Hash table and binary search tree structure: Extraction and population, memory usage and reserved memory over time.

(a) windows extraction rate, (b) memory usage over time, and (c) reserved memory over time.

As we can see, we have a slightly faster extraction rate than using a simple MAP structure (see figure 4.12(a)), but a slightly slower extraction rate than using a double MAP structure (see figure 4.15(a)). Memory usage over time is quite similar to the memory usage over time when using a simple MAP structure (see figure 4.12(b)), but is slightly less than the memory usage over time when using a double MAP structure (see figure 4.15(b)). For this reasons, a last implementation was tested. This implementation was a modification of the previous implementation, and aims to exploit the faster extraction rate of the double MAP structure instead of a simple MAP structure. A representation of this implementation is shown in figure 4.26.

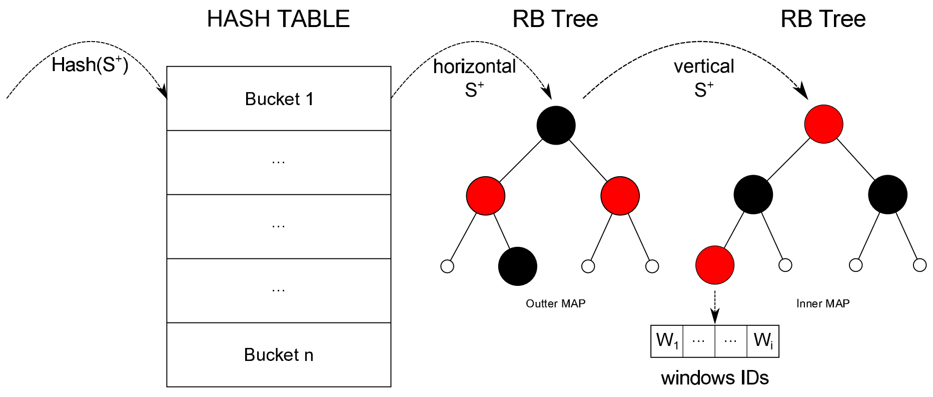
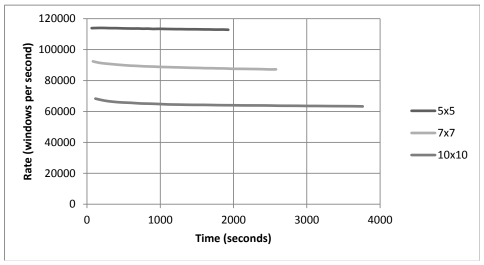
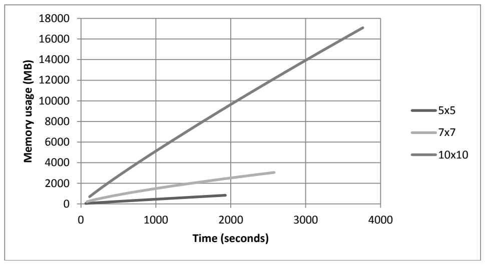


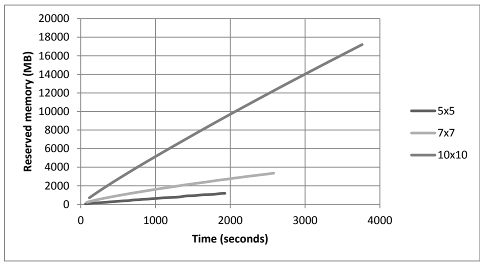
Figure 4.26: Representation of a structure using a hash table and two binary search trees.

Using this structure to store windows in the windows extraction process allows us to obtain extraction rates faster than any of the other structures. As figure 4.27(a) shows, when the size of the window of extraction is of 5×5, we obtain a extraction rate of around 113.000 windows per second. When the size of the window of extraction is of 7×7, we obtain a extraction rate of around 88.500 windows per second, and when the size of the window of extraction is of 10×10, we obtain a extraction rate of around 64.400 windows per second. The downside of this structure is the reserved use. Figure 4.27(c) show us that the reserved memory used by the structure when the size of the window of extraction is of 10×10 is significantly more than the other structures.



(a)





(b) (c)

Figure 4.27: Hash table and dual binary search tree structure: Extraction and population, memory usage and reserved memory over time.

(a) windows extraction rate, (b) memory usage over time, and (c) reserved memory over time.

In this section we have compared several data structures to store the information obtained from layouts in order to use it later. We use structures based on lists, binary search trees and hash tables, each one with different parameters such as initial reserved memory, number of buckets in the case of hash tables and more. The idea was to test the sensibility of the windows extraction rate and the memory usage for different data structures under different configurations. Due the results obtained, we use the last structure, i.e., the hash table with dual binary search trees as the structure to test the window-candidate retrieving quality of $\mathcal{S}^+$.

4.4 Searching patterns in the layout.

As is said in Chapter 3.4, when having a layout with their $\mathcal{S}^+$, and a pattern (e.g., a potential *hotspot*), it is possible to use $\mathcal{S}^+$ for selecting candidate windows of the layout that can contain that pattern. We can use $\mathcal{S}^+$ as a search key in the inverted index structure, which in our case is composed of a hash table and a dual-binary search tree structure. In the hash table, each bucket points to a red-black tree. In this red-black tree, each node identifies one value of the horizontal $\mathcal{S}^+$, and points to a second red-black tree. In this second red-black tree, each node identifies one value of the vertical $\mathcal{S}^+$, and points to a list of windows's IDs. These IDs identify all the windows whose $\mathcal{S}^+$ is formed by the horizontal $\mathcal{S}^+$ and the vertical $\mathcal{S}^+$ indicated by the red-black tree structure.

It is natural to think that the search process must retrieve all those windows whose $\mathcal{S}^+$ coincides with the $\mathcal{S}^+$ of a given pattern. This is partly true, but some circumstances are not fully covered. When one wants to identify the occurrence (or possible occurrence) of a certain geometrical pattern (e.g., a realization of a certain *range pattern*), one wants to be able to detect this geometrical pattern even when it can appears in the layout in various rotations (i.e., 0°, 90°, 180° and 270°), and even when each of these rotations appears mirrored. For these reasons, any $\mathcal{S}^+$ lookup must also looks for the previous mentioned spatial variations of this $\mathcal{S}^+$. This implies that each $\mathcal{S}^+$ lookup must be seen as 8 different lookups.

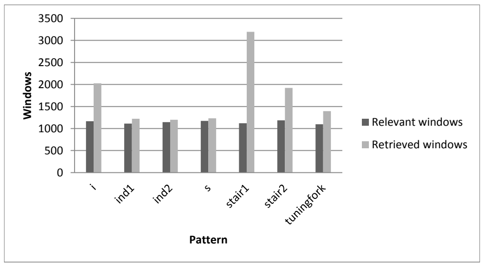
Searching for candidate windows takes the time needed for extracting the $\mathcal{S}^+$ from a *range pattern*, plus the time needed for finding the *signature* in the $\mathcal{S}^+$ indexing structure and then, traversing the list of windows with the same *signature*. The following section presents the experiments that check the quality of $\mathcal{S}^+$ as the search key to retrieve candidate windows. We used classical measures of *recall* and *precision* of information retrieval. *Recall* determines the number of relevant elements that are retrieved versus the total number of relevant elements, and *precision* determines the number of relevant elements that are retrieved versus the total number of retrieved elements. In simply words, *recall* serves to measure the number of false negatives, while *precision* serves to measure the number of false positives. Values close to 1 are desirable for both measurements. In our case, *recall* must be 1, because all patterns's realizations, at least, must be detected.

Additionally, as Chapter 3.3 indicates, various algorithms must be performed to extract $\mathcal{S}^+$ from a *range pattern*. We also present the time-cost measurements for each algorithm. We show the results obtained using size of windows of extraction of 10×10, that is the same size of the patterns used to build the layouts dataset, and size of windows of 5×5, which is a size smaller than the size of the patterns used to build the layouts.

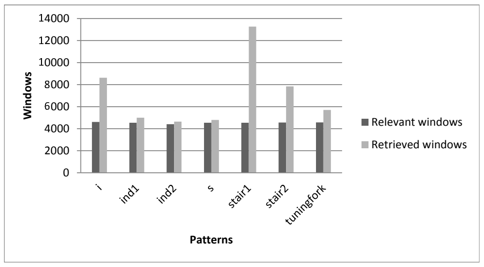
4.4.1 Searching with pattern and window of the same size.

When having a pattern realization of the same size of the windows of extraction, the retrieving of candidate windows that can contain that pattern realization is very simple. We must apply the algorithms presented in Chapter 3.3 and we obtain its $\mathcal{S}^+$. Then, we can use that $\mathcal{S}^+$ and its variations (0°, 90°, 180°, 270°, and mirroring) to search into the inverted index structure and retrieve all the windows that can contain the same geometrical configuration. To do so, for each $\mathcal{S}^+$ variation we calculate its hash value, and go through the hash table, then through the corresponding red-black tree that groups windows by its horizontal $\mathcal{S}^+$, and finally through the corresponding red-black tree that groups windows by its vertical $\mathcal{S}^+$. We then have a list of windows IDs that are candidates, i.e., can contain the given pattern realization. All the results indicate that the *recall* is always 100%, this is, $\mathcal{S}^+$ does not miss relevant elements.

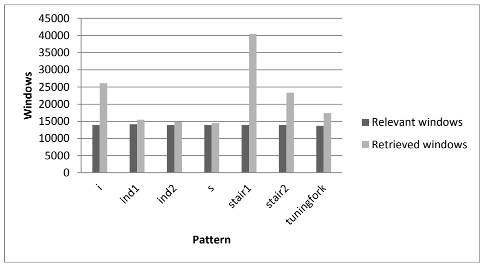
Figure 4.28 shows the precision of the candidate windows retrieved from three layouts of 4.000×4.000 (15.928.081 windows), 8.000×8.000 (63.856.081 windows), and 14.000×14.000 (195.748.081 windows), respectively. *Relevant windows* indicates the number of windows that actually contains a realization of the given pattern, while *Retrieved windows* indicates the number of windows retrieved by the filtering process using $\mathcal{S}^+$.



(a)



(b)



(c)

Figure 4.28: Precision for 7 different patterns in three different layouts.

(a) precision over a 4.000 × 4.000 layout, (b) precision over a 8.000 × 8.000 layout, and (c) precision over a 14.000 × 14.000 layout.

We can see that, the more complex is the pattern, the more precise is $\mathcal{S}^+$ to detect its occurrence in a layout, with precision that goes from 0,35 to 0,95 in the 4.000 × 4.000 layout, 0,34 to 0,95 in the 8.000 × 8.000 layout, and 0,34 to 0,95 in the 14.000 × 14.000 layout. The table 4.4 shows the access time, in seconds, to the corresponding candidate windows.

	4.000 × 4000	8.000 × 8.000	14.000 × 14.000
i	0,006666	0,029419	0,09443
ind1	0,00543	0,021374	0,067521
ind2	0,005721	0,020559	0,065822
s	0,006054	0,02133	0,065788
stair1	0,010377	0,046244	0,147539
stair2	0,007245	0,030893	0,097046
tuningfork	0,00612	0,025111	0,078211

Table 4.4: Accesing time cost table (in seconds).

As we can see, the hash with dual red-black search tree structure allows us to retrieve candidate windows with very low time-costs. However, it is also important to know the cost time used by each algorithm. Table 4.7 shows the time cost, in seconds, needed to run each algorithm over the 7 different patterns to retrieve candidate windows in the three layouts (Remember that the APMRP algorithm is invoked by the ENUM_DRG algorithm, and thus, its time cost is part of the cost of the ENUM_DRG algorithm).

	4.000×4.000	8.000×8.000	14.000×14.000
APMRP	8,32092	8,56205	8,65908
ENUM_DRG	9,0865	9,35278	9,45696
TOPOLOGICAL_ORDERING	0,01308	0,013478	0,013503
GET_SIG	0,6552	0,6392	0,6432

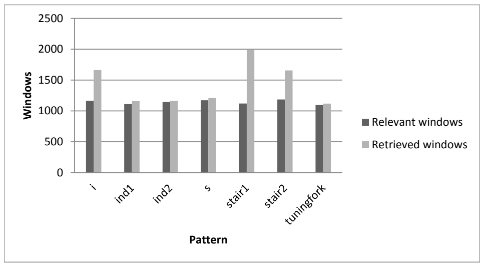
Table 4.5: Algorithms time cost table (in seconds).

We see that, even when the cost of some algorithms is $O(n^5)$, the execution time cost is negligible.

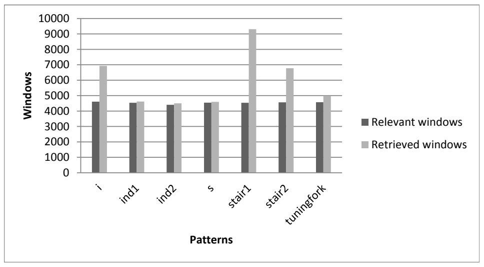
4.4.2 Searching with pattern and window of different size.

When we want to detect a pattern whose dimensions are larger than the dimension of the window of extraction, we can use a procedure like the one shown in Chapter 3.4. The algorithms performed are the same that the ones performed when the pattern and the window of extraction have the same dimensions, but the partition of a pattern into sub-patterns leads us to perform those algorithms more times. We want to compare the time-cost for the execution of the different algorithms, and the quality of $\mathcal{S}^+$ when retrieving candidate windows formed by a set of sub-windows (Here, a set of sub-windows conforms a bigger window that is the candidate window retrieved).

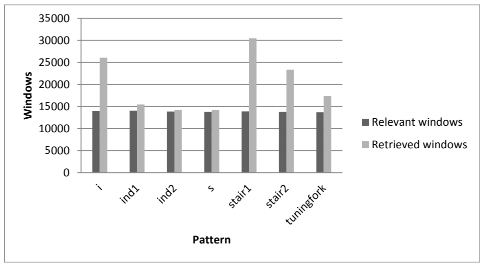
In this experiment, where we use a size of window of extraction of 5×5, the Recall is always 100%. Figure 4.29 shows the precision of the candidate windows retrieved from the same three layouts of 4.000×4.000 (15.928.081 windows), 8.000×8.000 (63.856.081), and 14.000×14.000 (195.748.081 windows), respectively. As in the previous results, *Relevant windows* indicates the number of windows that actually contains a realization of a given pattern, while *Retrieved windows* indicates the number of windows retrieved by the filtering process using $\mathcal{S}^+$.



(a)



(b)



(c)

Figure 4.29: Precision for 7 different patterns in three different layouts with patterns larger than the windows of extraction.

(a) precision over a 4.000×4.000 layout, (b) precision over a 8.000×8.000 layout, and (c) precision over a 14.000×14.000 layout.

We can see that, similar to the previous experiment, the more complex is the pattern, the more precise is $\mathcal{S}^+$ to detect its occurrence in a layout, with precision that goes from 0,56 to 0,98 in the 4.000×4.000 layout, 0,48 to 0,99 in the 8.000×8.000 layout, and 0,45 to 0,97 in the 14.000×14.000 layout. Focusing on the time cost of the procedure, the table 4.6 shows the access time, in seconds, to the corresponding candidate windows. We can note an noticeable increment in the time needed to access to the corresponding candidate windows. This increase is produced by the fact that we need to make a union between set of windows (one set for windows for each partition of the pattern), checking if the restriction of the IDs shown in Section 3.4 is satisfied in each case.

Finally, table 4.7 shows the time cost, in seconds, needed to run each algorithm over the 7 different patterns to retrieve candidate windows in the three layouts.

As we can see, the time cost associated with each algorithm increments, but it is still acceptable for all of them. The experiments have shown that it is possible to use $\mathcal{S}^+$ as *signature* for filtering out portions of the layout that do not contain a given

	4.000 × 4000	8.000 × 8.000	14.000 × 14.000
i	0,081063	0,357756	0,148338
ind1	0,046838	0,184368	0,582424
ind2	0, 0,065270	0,234556	0,750960
s	0,634062	0,2233986	0,689027
stair1	0,123036	0,5482970	0,749312
stair2	0,072844	0,310611	0,975741
tuningfork	0,050546	0,310611	0,645968

Table 4.6: Accesing time cost table.

	4.000 × 4.000	8.000 × 8.000	14.000 × 14.000
APMRP	84,7887	87,2458	89,9095
ENUM_DRG	117,1371	120,5698	121,912
TOPOLOGICAL_ORDERING	0,1686	0,1737	0,1740
GET_SIG	8,3380	8,1344	7,1853

Table 4.7: Algorithms time cost table.

pattern with a high quality measurement (*recall* and *precision*), while the time cost allows the use of the method in practice.

Chapter 5

Conclusions and future research directions.

We can conclude that the objectives of the thesis work, indicated at the beginning of this document are were achieved. A *signature* over rectangular configurations is defined. Our experiments has proved that it is possible to define a process that extracts this *signature* from an IC layout, requiering time and memory costs that can improve the current *hotspot* detection processes. Various indexing structures are introduced, each one with some advantages and disvantages. Empirically, we can conclude that using an indexing structure based on binary-search trees and hash tables has the better balance between time and memory costs. Also, a search method over the indexing structure is introduced, in order to obtain candidate areas that can contain certain geometrical pattern, even when the size of the candidate areas and the geometrical pattern is not the same, or the candidate area contains the geometrical pattern under rotations or mirroring.

The principal contribution of this thesis work is to define a process that can help IC fabs to reduce the costs, time and memory wise, of the *hotspot* detection process, allowing them to take advantage of the machine architecture that fabs use.

Future work can be focused on implement and optimize the *signature* extraction process using distributed architectures, and/or taking advantage of more hardware oriented optimizations. Even more, it is important to see that it is possible to benefit from parallel or group search over the indexed structure that stores the *signatures* of an IC design. However, for all the work and results obtained, we can conclude that using a *signature* as a *hotspot* detection support process can be very beneficial for the IC industry.

Appendix A

Range patterns.

```
Name = i;
Dir = all;
RectNum = 3;
LeftBry R0.l, R2.l;
BottomBry R2.b;
R0.r - R0.l is 10;
R1.r - R1.l is (2, 4);
R1.l - R0.l is (3, 4);
R2.r - R2.l is 10;
R2.l - R0.l is 0;
R0.t - R0.b is (2, 3);
R1.t - R1.b is (4, 6);
R0.b - R1.t is 0;
R2.t - R2.b is (2, 3);
R1.b - R2.t is 0;
```

Figure A.1: Pattern: i

```
Name = stair1
Dir = all;
RectNum = 3;
LeftBry R0.l;
BottomBry R0.b, R1.b;
R0.r - R0.l is (2, 3);
R1.r - R1.l is (7, 8);
R2.r - R2.l is (4, 6);
R1.l - R0.r is 0;
R2.l - R0.r is 0;
R0.t - R0.b is 10;
R1.t - R1.b is (2, 3);
R2.t - R2.b is (5, 6);
R0.b - R1.b is 0;
R2.b - R1.t is 0;
```

Figure A.3: Pattern: stair1

```
Name = stair2;
Dir = hor;
RectNum = 3;
LeftBry R0.l;
BottomBry R2.b;
R0.r - R0.l is (5, 7);
R1.r - R1.l is (5, 6);
R2.r - R2.l is (4, 5);
R1.l - R0.l is (2, 3);
R2.l - R1.l is (2, 3);
R0.t - R0.b is 2;
R1.t - R1.b is 2;
R2.t - R2.b is (2, 3);
R0.b - R1.t is 2;
R1.b - R2.t is (1, 2);
```

Figure A.2: Pattern: stair1

```
Name = tuningfork;
Dir = all;
RectNum = 4;
LeftBry R0.l;
BottomBry R0.b, R1.b, R2.b;
R0.r - R0.l is (2, 3);
R1.r - R1.l is (5, 6);
R2.r - R2.l is 2;
R3.r - R3.l is (2, 3);
R1.l - R0.r is 0;
R2.l - R1.r is 0;
R3.l - R1.l is (1, 2);
R0.t - R0.b is (6, 7);
R1.t - R1.b is (2, 3);
R2.t - R2.b is (6, 7);
R3.t - R3.b is (5, 6);
R0.b - R1.b is 0;
R0.b - R2.b is 0;
R3.b - R1.t is (1, 3);
```

Figure A.4: Pattern: tuningfork

```
Name = ind1;
Dir = all;
RectNum = 5;
LeftBry R0.l;
BottomBry R0.b;
R0.r - R0.l is (4, 5);
R1.r - R1.l is 2;
R2.r - R2.l is (4, 5);
R3.r - R3.l is (4, 5);
R4.r - R4.l is 2;
R0.r - R1.r is 0;
R2.l - R0.r is 1;
R3.r - R2.r is 0;
R3.l - R4.l is 0;
R0.t - R0.b is 2;
R1.t - R1.b is 8;
R2.t - R2.b is (2, 3);
R3.t - R3.b is 2;
R4.t - R4.b is (2, 3);
R1.b - R0.t is 0;
R0.t - R2.b is (0, 1);
R4.b - R3.t is 0;
R1.t - R4.t is 0;
```

Figure A.5: Pattern: ind1

```
Name = s;
Dir = all;
RectNum = 5;
LeftBry R3.l;
BottomBry R0.b;
R0.r - R0.l is (5, 6);
R1.r - R1.l is 2;
R2.r - R2.l is 6;
R3.r - R3.l is 2;
R4.r - R4.l is (5, 6);
R1.l - R0.r is 0;
R2.r - R1.l is 0;
R2.l - R3.r is 0;
R3.r - R4.l is 0;
R0.t - R0.b is 2;
R1.t - R1.b is (5, 6);
R2.t - R2.b is 2;
R3.t - R3.b is (4, 5);
R4.t - R4.b is 2;
R0.t - R1.b is (1, 2);
R1.t - R2.t is 0;
R2.t - R3.b is 1;
R4.t - R3.t is (0, 1);
```

Figure A.7: Pattern: s

```
Name = ind2;
Dir = ver;
RectNum = 5;
LeftBry R0.l;
BottomBry R0.b, R3.b;
R0.r - R0.l is 2;
R1.r - R1.l is 2;
R2.r - R2.l is 2;
R3.r - R3.l is 2;
R4.r - R4.l is 2;
R1.l - R0.r is 0;
R2.l - R1.r is 0;
R3.l - R0.r is 4;
R4.l - R3.r is 0;
R0.t - R0.b is (4, 5);
R1.t - R1.b is (4, 5);
R2.t - R2.b is (4, 5);
R3.t - R3.b is 4;
R4.t - R4.b is 5;
R0.t - R1.b is (2, 3);
R1.t - R2.b is (1, 2);
R0.b - R3.b is 0;
R3.t - R4.b is 2;
```

Figure A.6: Pattern: ind2

Bibliography

[1] Cristian Andrades. *Una optimización del reconocimiento de hotspots en la manufactura de circuitos integrados basado en la extracción de signatures.* 2013.

[2] J.-M. Lin.; Y.-W. Chang. *TCG: A Transitive closure graph based representation for non-slicing floorplans.* In In Proceedings of DAC, pp. 764–769., 2001.

[3] C. Andrades; M. Andrea Rodríguez.; Charles C. Chiang. *Signature indexing of design layouts for hotspot detection.* In Proceedings of DATE'14, pp. 1–6., 2014.

[4] Yen-Ting Yu.; Ya-Chung Chan.; Subarna Sinha.; Iris Hui-Ru Jiang.; Charles Chiang. *Accurate process-hotspot detection using critical design rule extraction.* In The 49th Annual Design Automation Conference 2012 (DAC'12), pages 1167-1172. ACM., 2012.

[5] A. B. Kahng et al. *Fast dual graph based hotspot detection.* In Proceedings of SPIE, vol. 6349, pp. 628–635., 2006.

[6] H. Yao.; S. Sinha et al. *Efficient Process-Hotspot Detection Using Range Pattern Matching.* In Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design, pp. 625 – 632., 2006.

[7] T. Kotani.; H. Ichikawa et al. *Lithography Simulation System for Total CD Control from Design to Manufacturing.* In Proceedings of SPIE, Vol. 5756, pp. 219 – 229., 2005.

[8] T. Kotani.; S. Kyoh et al. *Development of Hot Spot Fixer (HSF).* In Proceedings of SPIE, Vol 6156, 61560H 1-8., 2006.

[9] T. Kotano.; S. Tanaka et al. *Yield Enhanced Layout Generation by New Design for Manufacturability(DFM) Flow.* In Proceedings of SPIE, Vol. 5379, pp. 128-138., 2004.

[10] Jr. F. G. Pikus.; T. W. Collins. *Topological pattern matching.* US Patent Application 2010/018594 A1., 2010.

[11] M. Cote.; P. Hurat. *Standard Cell Printability Grading and Hot Spot Detection.* In Proceedings of Sixth International Symposium on Quality of Electronic Design, Vol. 113, pp. 264-269., 2005.

[12] Saint Christopher.; Saint Judy. *IC Layout Basics: A Practical Guide.* McGraw-Hill., 2001.

[13] Ning. Ma. *Automatic IC Hotspot Classification and Detection using Pattern-Based Clustering.* University of California, Berkeley., 2009.

[14] P. Brooker.; H. Koop.; H. Marquardt. *Using DFM Hotspot Analysis for Predicting Resist Patterns.* Solid State Technology, Vol. 48 (12), pp. 47 – 49., 2005.

[15] Frank E. Gennari.; Ya-Chieh Lai.; Matthew W. Moskewicz.; Michael C. Lam.; Gregory R. McIntyre. *Fast pattern matching.* US Patent, US 7818707 B1., 2010.

[16] D.Z. Mitra, J.; Peng Yu.; Pan. *RADAR: RET-aware detailed routing using fast lithography simulations.* In Proceedings of the 42st Design Automation Conference 2005., 2004.

[17] A. A. Y. Mustafa. *Fuzzy shape matching with boundary signatures.* Pattern Recognition Letters, vol. 23, no. 12, pp. 1473–1482., 2002.

[18] Duo Ding.; Andres J. Torres.; Fedor G. Pikus.; David Z. Pan. *High performance lithographic hotspot detection using hierarchically refined machine learning.* In Proceedings of the 16th Asia South Pacific Design Automation Conference, ASP-DAC, pages 775-780. IEEE., 2011.

[19] Duo Ding.; Andres J. Torres.; Fedor G. Pikus.; David Z. Pan. *High performance lithography hotspot detection with successively refined pattern identifications and machine learning.* IEEE Trans. on CAD of Integrated Circuits and Systems, 30(11):1621-1634., 2011.

[20] Duo Ding.; Xiang Wu.; Joydeep Ghosh.; David Z. Pan. *Machine learning based lithographic hotspot detection with critical-feature extraction and classification.* In IEEE International Conference on IC Design and Technology, ICICDT, pages 219-222., 2009.

[21] J. Chanussot.; I. Nyström.; N. Sladoje. *Shape signatures of fuzzy star-shaped sets based on distance from the centroid.* Pattern Recognition Letters, vol. 26, no. 6, pp. 735–746., 2005.

[22] Li-Da Huang.; Martin D. F. Wong. *Optical Proximity Correction (OPC)-Friendly Maze Routing.* In Proceedings of the 41st Design Automation Conference 2004., 2004.

[23] J.-Y. Wuu. *Rapid layout pattern classification.* In Proceeding of ASP-DAC, pp. 781–786., 2011.