POLYGON PACKING APPROACH TO DISCONNECTED GRAPH LAYOUT

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER

ENGINEERING

AND INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

by

Cihad Başköy

January, 2003

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

_____

Asst. Prof. Dr. Uğur Doğrusöz (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

_____

Assoc. Prof. Dr. Özgür Ulusoy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

_____

Asst. Prof. Dr. Uğur Güdükbay

Approved for the Institute of Engineering and Science

_____

Prof. Dr. Mehmet Baray

Director of Institute of Engineering and Science

ABSTRACT

POLYGON PACKING APPROACH TO DISCONNECTED GRAPH LAYOUT

Cihad Başköy
M.S. in Computer Engineering
Supervisor: Asst. Prof. Uğur Doğrusöz
January, 2003

Graph layout has become an important area of research in Computer Science for the last couple of decades. There is a wide range of applications for graph layout including data structures, databases, software engineering, VLSI technology, electrical engineering, production planning, chemistry, and biology. Most layout algorithms assume the graph to be connected. However, most graphs are disconnected and a method for putting the disconnected graph objects together is needed.

Two-dimensional packing algorithms have wide area of application such as in the steel and textile industry. In steel industry, problems frequently occur when the need to stamp polygonal figures from a rectangular board arises. In the textile industry, similar problems exist. The aim is same: to maximize the use of the contiguous remainder of the board.

Recently, two-dimensional packing has also been used in disconnected graph layout yielding algorithms that 'tile' the disconnected graph objects, which are represented by rectangles. These algorithms are also required to respect the specified aspect ratio for the final layout. A more recent approach to disconnected graph layout has been the use of polyominoes for representing the graph objects resulting in more accurate packings at the cost of increased execution times.

In this thesis, we use polygons for a more accurate representation of graph objects and present new algorithms for disconnected graph layout. Specifically, we apply the No-Fit Polygon approach in two-dimensional packing to disconnected graph layout. We present and analyze the graph layouts resulting from our new approach and contrast the new approach with previous ones.

Keywords: Graph Layout, Disconnected Graph Layout, Two-Dimensional Packing.

ÖZET

AYRIŞIK ÇİZGE YERLEŞTİRİMİNDE POLİGON PAKETLEME YAKLAŞIMI

Cihad Baskoy
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yrd. Doç. Dr. Uğur Doğrusöz
Ocak, 2003

Çizge yerleştirimi konusu, Bilgisayar Bilimlerinde son birkaç on yıl içerisinde önem kazanmıştır. Kullanım alanları oldukça geniş olmak ile beraber, veri yapıları, veritabanları, yazılım mühendisliği, VLSI teknolojileri, elektrik mühendisliği, üretim planlaması, kimya ve biyoloji örnek olarak verilebilir. Yerleştirme algoritmalarının bir çoğu, çizgenin bağlaşık olduğunu varsayarak işlem yapmaktadır. Bununla birlikte, karşılaşılan çizgelerin ayrık olması durumunda kullanılabilecek algoritmalara ihtiyaç duyulmaktadır. İki boyutlu kap içerisine paketleme problemleri tekstil ve çelik endüstrilerinde geniş uygulama alanına sahiptir. Örneğin çelik endüstrisinde, dikdörtgen plakalar üzerinde poligonsal şekillerin işaretlenmesi ve kesilmesi önemli bir problemdir. Bütün bu uygulamalardaki ana amaç, kullanılmayan alanın en aza indirimesi olarak özetlenebilir.

Son zamanlarda, iki boyutlu paketleme algoritmaları, ayrık çizge yerleştirimi araştırmalarında, dikdörtgenler şeklinde tanımlanan ve bağlantılı olmayan çizge elemanlarını yüzey üzerine kaplamak için kullanılmaya başlanmıştır. Bu algoritmaların gereklerinde birisi, oluşturulacak yerleştirimde önceden tanımlanan boyut oranının korunmasıdır. Çizge yerleştirimi konusunda, elemanların polyomino'lar kullanılarak tanımlanması, hesaplama zamanlarını artırmakla birlikte daha doğru sonuçlar elde edilmesini sağlayan, yeni bir yaklaşımdır. Çizim performansı ve hesaplama zamanı arasındaki denge önemlidir.

Bu çalışmada, çizge elemanlarının detaylı olarak tanımlanabilmesi için poligonlar kullanılmakta ve ayrık çizge yerleştirimini gerçekleştiren yeni bir algoritma sunulmaktadır. Özetle, ayrık çizge yerleştirimlerinin iki boyutlu paketlenmesi için No-Fit poligon yaklaşımı uygulanmaktadır. Bu yeni yaklaşım kullanılarak elde edilen sonuçlar sunulmakta ve değerlendirilmektedir. Ayrıca bu sonuçların, önceki yaklaşımların sonuçları karşılaştırılması yapılmaktadır.

Anahtar Sözcükler: Çizge yerleştirimi, Ayrık çizge yerleştirimi, İki boyutlu paketleme.

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Uğur Doğrusöz, from whom I have learned a lot, due to his supervision, suggestions, support, and patience during this research.

I would like to thank to my friend Emre Erdoğan for his improvements and ideas on this research.

And finally, I would like to thank to my wife for her support.

Contents

Table of Contents

List of Figures

1. Introduction

1.1 Overview

Graph drawings model and help us visualize the complex information in a system of discrete objects and their relationship. Graph layout is the automatic positioning of the nodes and edges of a graph in order to produce an aesthetically pleasing drawing that is easy to comprehend.

Many graph layout and editing systems have been developed in the past [1,2]. One essential aspect that has not been addressed sufficiently, is the layout of disconnected graph; that is, the placement of the components of a disconnected graph. Disconnected graphs occur rather frequently in real life applications either during the construction of the nature of application (Figure 1.1).

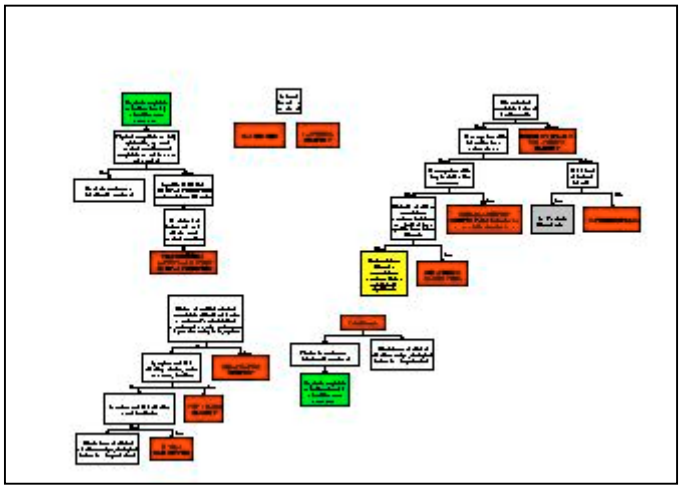

Figure 1.1: An example of a disconnected graph

Most graph layout algorithms assume a graph to be connected and try to minimize the area needed for the resulting drawing. No matter how effective such an algorithm is, the space wasted overall could be arbitrarily large if the relative locations of disconnected objects of a graph are chosen by a naïve, inefficient method.

Another key parameter here is the aspect ratio of the region (e.g., a window) within which the graph is to be displayed (Figure 1.2). When displaying a graph, the larger the wasted space is, the less visible objects will be, making the visualization process more difficult. Thus, a disconnected graph layout algorithm must strive for a packing of disconnected objects which respects the aspect ratio of the region in which it is to be displayed.

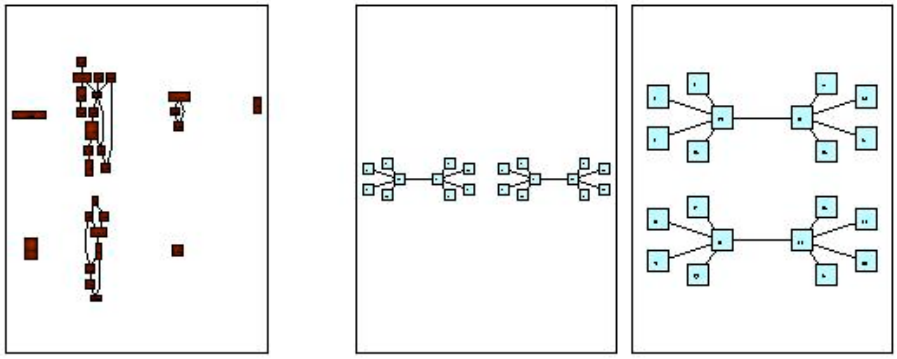

Figure 1.2 : How a naive disconnected graph layout algorithm can make inefficient use of the are (**left**), and why the aspect ratio of the region in which the graph is to be drawn should be taken into account during dixconnect graph layout (**middle and right**).

Packing problems have been topics of extensive study in both operations research and computer science because of their extremely wide application areas and their great theoretical challenge [3]. Some of the well-known packing problems include:

- **One-dimensional Bin Packing** The problem of putting variable sized items (the sizes are represented as integers) into bins of the same capacity and minimizing the number of bins used, subject to the condition that the total size in each bin does not exceed the capacity.

- **Cutting Stock** An extension of one-dimensional bin packing problem. Instead of having the same capacity, the bins have *k* different capacities, where *k* is fixed. There are unlimited supplies of bins for each capacity. Bins of the same capacity are assigned the same cost. The objective is to minimize the total cost while packing the items into these different sized bins. This problem is also called *variable sized bin packing* when the cost of a bin is proportional to its capacity.

- **Two-dimensional Bin Packing** The problem of packing a set of rectangles of different height and width into rectangular bins such that the rectangles do not overlap with each other. The bins are of the same width and height. Usually there are additional restrictions on the orientation of the rectangles, such as such as the edges of a rectangle must be parallel to those of the bin that contains it and no rotation of a rectangle is allowed. The objective is to use the minimal number of bins to pack all the rectangles. Variations of the problem is called :

    - **Strip Packing** Instead of using many bins of the same size, a bin of fixed width and unlimited length is used and the objective is to pack all the rectangles using minimum length.

    - **Tiling Packing** Elimination of the to 'guess' the right size strip by maintaining a bin whose width dynamically changes (i.e. increases). Tiling packing is also called as *Strip-Packing with Variable Width Strip*.

    - **Alternate-Bisection Packing** A method based on an alternate-bisection technique used in floorplanning in integrated circuit layout. This divide-and-conquer method works by bisecting disconnected objects of a graph alternately.

- **Two-dimensional Packing** Generalizations of two-dimensional bin packing in which the objects being packed can be non-rectangular and the restrictions on the orientation can also be lifted. Variations of the problem is called :

    - **Polyomino Packing** Each graph object represented by a polyomino and by using this polyomino, placing each object one by one, finding the optimal place for the new object, one at a time, with respect to the already placed ones.

    - **No-Fit Polygon Packing** Placing each object one by one by using No-Fit polygon concept that is determining all the arrangements that two arbitrary objects may assume such that the shapes do not overlap. The objects are

assumed to have fixed absolute orientations but are free to move anywhere on the 2-D plane. [4,5,6]

A specified aspect ratio can be applied to two-dimensional algorithms for the layout of disconnected graphs for based on *strip packing*, *tiling packing*, *alternate-bisection packing*, *polyomino packing* and *no-fit polygon packing*.

The contribution of this thesis is to contrast no-fit polygon packing with tiling, alternate-bisection and polyomino packings. Using no-fit polygon packing at two-dimensional packing is an important approach to contrast the graph layout performance and time performance against tiling, alternate-bisection and polyomino packings since these algorithms simulate polygons as rectangles or polyominoes. No-fit polygon packing approach is direct calculation of the layout of the graph with given polygons, without describing them to something else. Expected result from this contrast is no-fit polygon packing shall have better graph-layout performance, but worse time-complexity against other approaches.

1.2 Definitions and Basics

From now on, the terms ‘graph object’, or simply ‘object; are used interchangeably to denote a component or an isolated node of the graph to be laid out.

The tightest rectangle bounding the drawing of a graph object or the entire graph is said to be its *bounding rectangle*. The aspect ratio of a rectangle $R = (W, H)$ is equal to W/H.

Most layout algorithms represent graph objects with either points or rectangles in the plane. A polyomino is a geometric figure formed by joining unit squares at the edges. For polyomino packing algorithm, polyominoes are used to represent graph objects. (Figure 1.3)

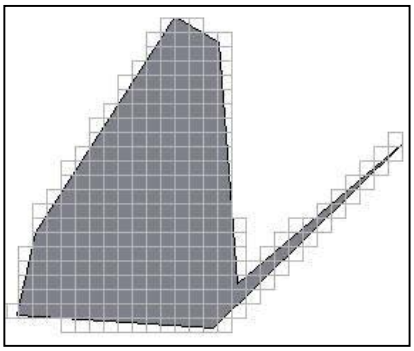

Figure 1.3 : A graph object represented by polyominoes.

No-fit polygon (NFP) is all the arrangements that two polygons may assume without overlapping. Given two polygons A and B such that the position of A, the orientation of A, and the orientation of B are all fixed, then the NFP of B relative to A completely describes all those positions where the reference point of B may be placed in order to have B touching A without overlapping it. (Figure 1.4) [7]

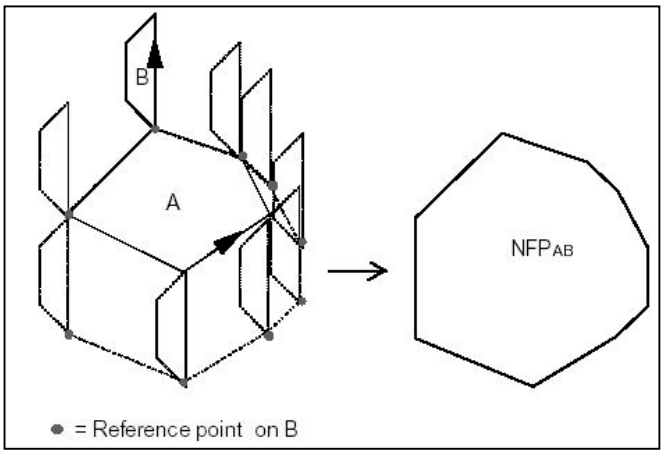

Figure 1.4 : The locus of the reference point on B maps out the nofit polygon as B traces around A.

2. Related Work

The related work for constructing a graph layout from objects in our thesis can be divided into four parts: Strip packing, Tiling packing, Alternate-bisection packing and, Polyomino packing. Technical details of these packing methods will be given at this chapter.

2.1 Strip Packing

In strip packing, given list of $n \geq 1$ rectangles $L_n = (R_1, R_2, \ldots, R_n)$, each having dimensions (W_i, H_i), are to be packed into a semi-infinite strip of unit width so that :

(i) Rectangles do not overlap each other or the edges of the strip,
(ii) The rectangles must be packed with their sides parallel to the edges of the strip (no rotations from the given orientations are allowed),
(iii) The height of the packing is minimized. [2]

It is assumed that the rectangles in the list L_n are ordered by decreasing (actually, nonincreasing) height, and they pack the rectangles in the order given by L_n so as to form a sequence of *levels*. A packing of L_n is a *level packing* if there exists a sequence of horizontal cuts at heights $C_0 = 0 < C_1 < C_2 < \ldots < C_k$, $1 \leq k \leq n$, such that:

(i) First level is simply the bottom of the bin,
(ii) No cut passes through any rectangle,
(iii) The total width of rectangles between C_{i-1} and C_i, $1 \leq i \leq k$, is at most 1, the strip width.

An algorithm that produces only level packings is called a *level algorithm* [8]. Following two level algorithms are most popular ones:

- **Next-Fit Decreasing-Height (NFDH)** With this algorithm, rectangles are packed left-justified on a level until there is insufficient space at the right to accommodate the next rectangle. At that point, the next level is defined, packing on the current level is discontinues, and packing proceeds on the new level.
- **First-Fit Decreasing-Height (FFDH)** At any point in the packing sequence, the next rectangle to be packed is placed left justified on the first (i.e., lowest) level on which it will fit. If none of the current levels will accommodate this rectangle, a new level is started as in the NFDH algorithm.

For L an arbitrary list of rectangles, all assumed to have width no more than 1, let OPT(L) denote the minimum possible bin height within which the rectangles in L can be packed, and let A(L) denote the height actually used by a particular algorithm when applied to L. The *wasted space* $WS^A(L)$ is the unoccupied area of the packing:

$$WS^A(L) = A(L) - \sum_{i=1}^{n} W_i H_i$$

Similarly, fullness of packing $F^A(L)$ expresses, in percentage, how effectively the area is used by the packing algorithm [2]:

$$F^A(L) = 100 \cdot \left(\sum_{i=1}^{n} W_i H_i \right) / A(L)$$

The results in [8,9] are concerned primarily with demonstrating *absolute performance bounds* for various algorithms A, i.e., bounds of the form

$$A(L) \leq \alpha \cdot OPT(L) + \beta$$

for all lists. Moreover, by modifying the above algorithms, it is stated at [10] that asymptotic bound α may be decreased to 5/4 and this bound is tight.

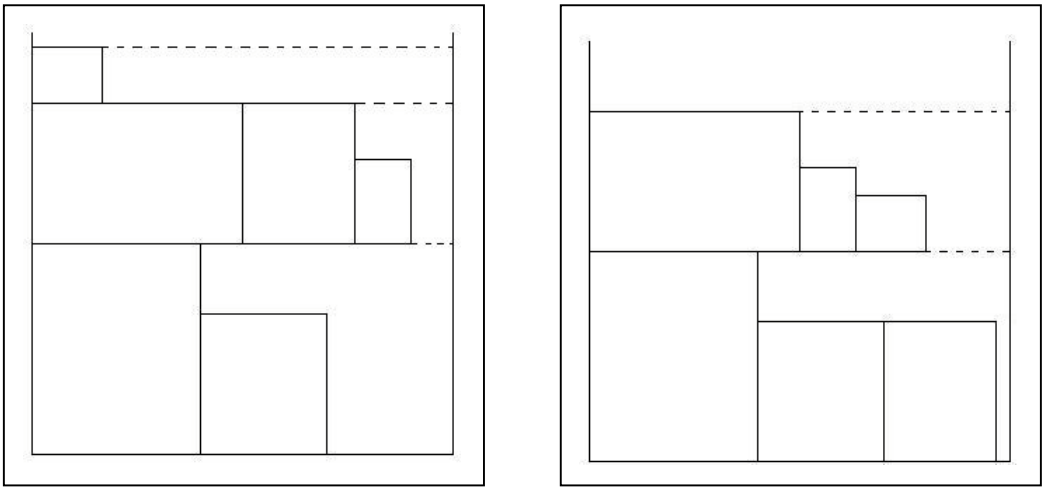

Figure 2.1 : NFDH (**left**) and FFDH (**right**) algorithms applied to the list of rectangles L. NFDH(L) = 3/4 and FFDH(L) = 13/20.

Figure 2.1 shows the results of applying the two packing rules to the same list. The essential difference between them is that wheras FFDH can always return to a previous level for packing a new rectangle, NFDH always place subsequent rectangles at or above the current level.

When concerning aspect ratio, the width of the strip based on the desired aspect ratio is calculated by using the theoratical performance of the strip-packing algorithm. In other words, the width of the strip should be such that the resulting packing yields an aspect ratio close to desired one. Notice that the desired aspect ratio will be achieved only when the worst-case performance is hit; otherwise, the aspect ratio will be larger than the desired aspect ratio. Figure 2.2 illustrates how the size of the strip can be set or the dimensions of the rectangles can be scaled to obtain an aspect ratio closer to the desired one upon application of the FFDH algorithm with an example. Notice that both kinds of adjustments result in the same packing.

The algorithm is of $O(n \log n)$ time complexity.

Figure 2.2 : Choice of the strip width or the scaling factor for rectangle dimensions affect the aspect ratio of the resulting packing. The rectangles on the left are strip-packed using the FFDH algorithm. In the middle, the same set of rectangles are again packed with FFDH but using a strip that is not as wide. On the right are same rectangles, but this time scaled to be larger, packed with FFDH using a strip same width as the one left.

2.2 Tiling Packing

The tiling packing eliminates the need to 'guess' the right size strip by maintaining a bin whose width *dynamically* changes (i.e., increases) [1,2,3]. The algorithm starts by creating an initial level and placing the first rectangle in this level. It proceeds by determining whether the next rectangle in line should be added to one of the existing levels (the one, which is the least utilized at moment) or to a newly created level. The rectangle is tiled on one of the existing levels if there is enough room. Otherwise, a decision is made on whether the current strip width should be enlarged or a new level should be formed to keep the aspect ratio closer to the desired one.

In general, the tiling algorithm does not assume any particular ordering of the objects. However, experiments show that when graph objects are sorted in nonincreasing height, most compact drawings are obtained. Notice that when objects are processed in order of nonincreasing height, the algorithm turns into a variation of a strip-packing algorithm, FFDH to be more specific, where the strip width is dynamically increased as necessary to better fulfill the aspect ration constraint.

Figure 2.3 shows an example application of the tiling algorithm. The rectangles are processed in the ascending order of their labels in this particular example. For instance when placing object '2', an existing level is used, and the bin width is enlarged; whereas, when placing object '3', a new level is rather created.

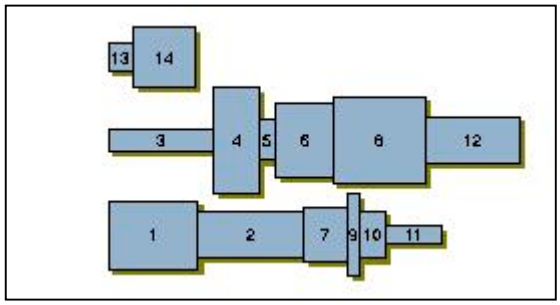

Figure 2.3 : An example of the application of the tiling algorithm

Figure 2.4 illustrates an example of tiling packing algorithm described above with 30-40 graph objects. Graph layout at the left has desired aspect ratio 2.0 and actual aspect ratio 1.9807693, and by the same graph objects, the graph layout at the right has desired aspect ratio 0.5 and actual aspect ratio 0.49834436.

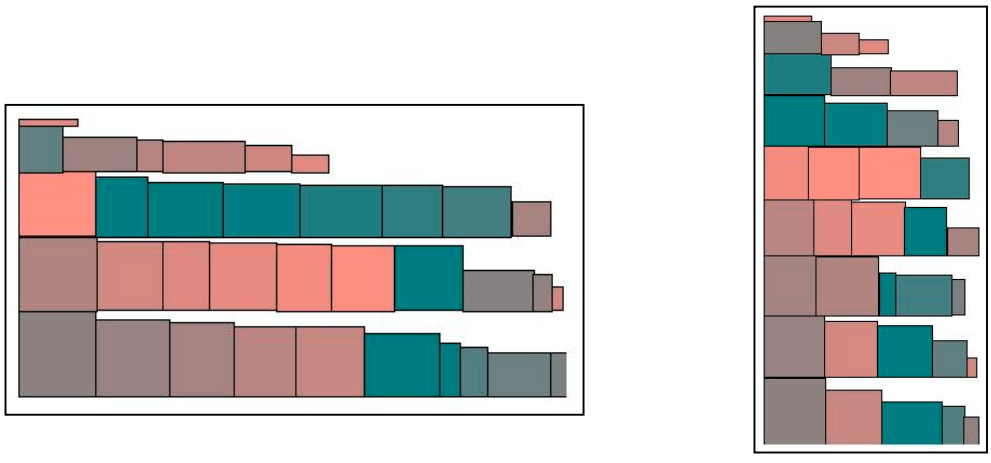

Figure 2.4 : An example of tiling packing produced by this algorithm with desired aspect ratio 2.0 (**left**) and desired aspect ratio 0.5 (**right**) of same graph objects.

The algorithm is of $O(n \log n)$ time complexity [1].

2.3 Alternate-Bisection Packing

This is a divide-and-conquer method works by bisecting the disconnected objects of a graph alternately as follows [1,2]. The objects are bipartioned using a metric such as total area and objects in each partition are recursively laid out. The recursion continues until a partition consists of a small, constant number of objects (e.g, one) whose optimal layout becomes easy if not trivial. At the end of each recursive step, when placing the two embedded partitions relatively, the orientation is alternated. For instance, the last step would place the two already positioned partitions side by side (horizontally) if the four partitions in the previous step were placed one on top of the other (vertically) pairwise.

Figure 2.5 illustrates this method with an example.

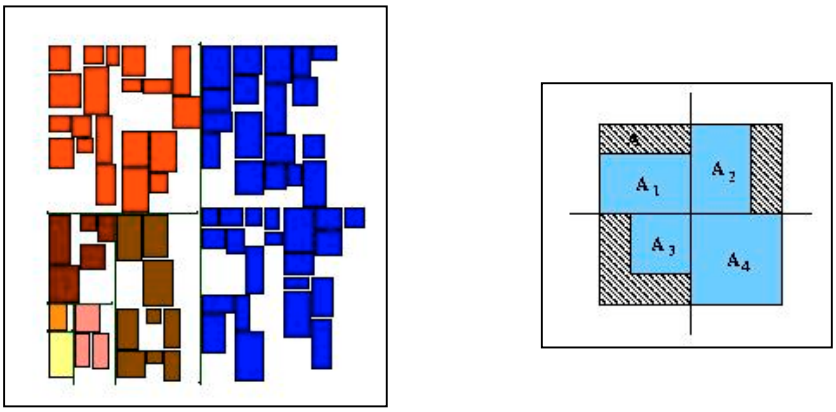

Figure 2.5: An example of the application of alternate-bisection packing method; on one thread of the recursion, alternately partioned objects are shown with seperating lines and colors (**left**). An illustration of how the four partitions of the objects recursively packed are put together; $A_i = W_i \cdot H_i$, $i = 1,\ldots,4$, and A denote the areas of the four partitions, and the area of the combined packing, respectively (**right**).

Total area wasted by the algorithm for n objects W(n), is roughly $O(n^{1.41})$ [1,2], which is quite inefficient. However, when simple alternating ordered one-dimensional packings are applied in each recursive step, much more compact results are obtained as the experimental results show. Figure 2.6 shows the same set of objects packed without and with the use of ordered one-dimensional packings; clearly make a positive difference.

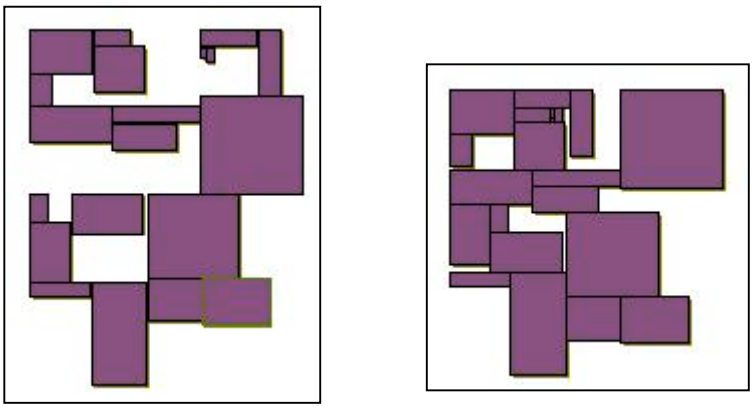

Figure 2.6: The same set of objects packed with alternate-bisection; ordered one dimensional packing is not used in the left one.

2.4 Polyomino Packing

In this approach, each graph object is represented by a polyomino. A polyomino is defined as a finite set of $k \geq 1$ cells of infinite planar square grid G that are fully or partially covered by the drawing of the object. If the case that an object is placed completely inside another one is not desirable, the definition can be modified and the uncovered grid cells that are completely bounded by the covered ones can be included as well.

According to [1], given a set of polyominoes P_i, $1 \leq i \leq n$, packing them into a minimum area is NP-hard even if the polyominoes are restricted to rectangles. The heuristic algorithm for polyomino packing given at [1] is a greedy algorithm: it places the objects one-by-one, finding the optimal place for the new object, one at a time, with respect to the already placed ones. The optimal place for a polyomino is simply calculated as the grid cell G_{xy} located at (x,y) where the function $\max(|x|,|y|)$ is minimized over all grid cells. The cost function defines the order in which the cells are examined and this order is the same for all polyominoes.

To find the best place for graph object represented by polyominoes, algorithm looks sequentially through all cells in the increasing order of the cost function defined as $\max(|x|,|y|)$. If an available spot is found, it is placed there and the corresponding grid cells are

marked as occupied. To find intersections, algorithm go through all polyomino cells and test each of the polyomino can be placed in a free grid cell.

The pseudo code of algorithm shall be given as:

```
algorithm PolyominoPacking (Pi, 1 ≤ i ≤ n)
(1)   sort Pi, 1 ≤ i ≤ n in the order of nonincreasing size
(2)   initialize the grid G using the sizes of Pi, 1 ≤ i ≤ n
(3)   foreach polyomino Pi do
(4)         calculate (x,y) such that the cost function is minimized
(5)         while cannot place Pi in G centered (x,y) do
(6)               calculate next (x,y) using the cost function
(7)         end while
(8)         mark the cells in G covered by Pi as occupied
(9)   end foreach
```

Figure 2.7 illustrates an example drawing produced by using above algorithm with cost function equals to minimizing absolute value of aspect ratio minus desired aspect ratio by 30-40 graph objects.

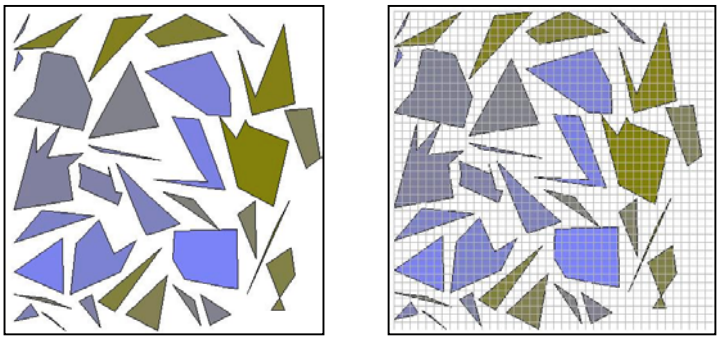

Figure 2.7 : An example of packing produced by this algorithm with desired aspect ratio 1.0. The resulting graph layout with calculated grids are also given (**right**).

2.4.1 Parameters of Polyomino Packing

Grid step (i.e., grid size) is the most significant parameter of polyomino packing. To guarantee the avarage polyomino size s is constant, the grid step l is calculated from the following quadratic equation:

$$cl^2 - Pl - A = 0$$

where A is the avarage area of the polyomino bounding rectangles, P is the avarage half perimeter of the polyomino bounding rectangles, and c is a constant to specify the approximation quality [1]. Then the avarage polyomino size is $s = O(c)$ and the total area of all polyominoes is $O(n \cdot s)$.

The complexity of the algorithm is $O(n^2 \cdot s^2)$. Since c and consequently s are constants, this yields an $O(n^2)$ time overall [1].

To satisfy the desired aspect ratio, simply take desired aspect ratio as the unit grid step in x direction and 1 as the unit step in y direction.

2.5 Comparison of Packing Algorithms

In [1], the comparison of tiling, alternate-bisection and polyomino packing approaches is given. Graphs that contained up to a thousand disconnected objects were user where each object was assumed to be a polygon with random number of corners in [3…8]., each with random integer coordinates in [1…100], all independent and uniformly distributed. The value for the approximation quality constant c (a parameter for polyomino packing) was taken to be 100. For the tiling and alternate-bisection methods the tighest rectangles bounding these polygons were used, whereas for polyomino packing the smallest polyomino bounding the polygons were used. Figure 2.8 show a sample set of drawings produced by these methods for the same set of objects. The performance comparison is presented in Figure 2.9.

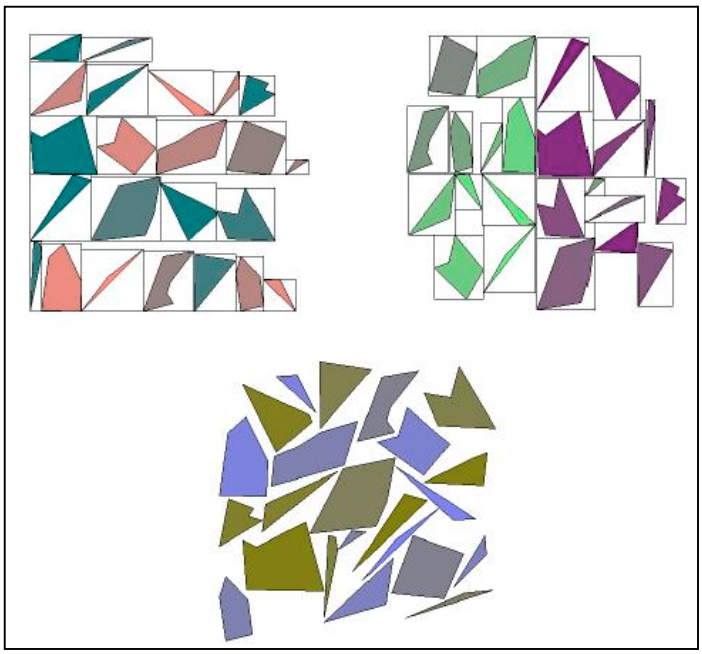

Figure 2.8 : A sample from the random set of objects laid out with three methods: tiling (**upper left**), alternate-bisection (**upper right**), and polyomino (**bottom**).

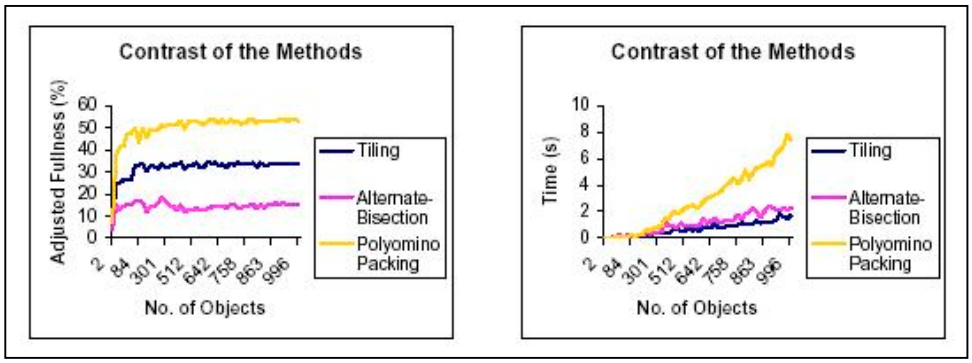

Figure 2.9 : Comparison of the polyomino approach with previous ones.

Polyomino packing approach results in much more compact drawings. In terms of execution time, it is slower but still easily within acceptable bounds given the fact that it is highly rare that a graph contains more than a few hundred disconnected objects [1].

3. No-Fit Polygon Packing Approach

Polyomino packing yields good results for disconnected graph layout if polygons are used for representing graph objects. Whether this method wastes considerably amount of area since the grids of polyomino may not be covered by a polygon fully. Therefore, if a polygon is not defined by polyominoes (i.e., dividing the polygon into grids), a better performance can be reached.

Recently, in [4], No-Fit Polygon (NFP) is used for two-dimensional bin packing problem. In this research, only convex polygons are considered but NFP brings out a solution for two-dimensional bin packing problem. In this method, pieces are placed into a bin one at a time. The location of the next piece is calculated using the NFP. Once the best placement is found, the piece is added to the partial solution moving on to the next piece is placed [4,5,6,7].

3.1 No-Fit Polygon

The term NFP was firstly introduced in [11]. NFP determines all arrangements that two arbitrary polygons may assume such that the shapes do not overlap or that they cannot be moved closer together without intersecting. The NFP of two polygons A and B, denoted as NFP_{AB} is the polygon that results from a sliding operation in which A and B have specific roles. The first polygon in the subscript is defined as the fixed (constant) polygon whose origin is assumed to be at point (0,0). The second polygon is called the tracing (rotating) polygon and is moved to perform the sliding operation [7,13].

Given a physical representation of the two objects, NFP_{AB} can be obtained by placing B in a touching position with A and marking the locus of a reference point on B as it traces around the boundary of A. The tracing motion of B is performed in such a way that B does not rotate

and *A* and *B* always touch, but never overlap. The locus of the reference point forms a closed path that is NFP_{AB}. Figure 1.4 illustrates this tracing movement [7].

If the roles are reversed then the resulting NFP, NFP_{BA}, is NFP_{AB}, rotated by 180°. The relevant property of NFP_{AB} with respect to the interaction between *A* and *B* is as follows: if *A* is placed at (0,0) and *B* is positioned with its reference point inside NFP_{AB}, then *A* and *B* intersect; and if *B* is positioned with its reference point on the boundary of NFP_{AB}, then *A* and *B* touch. Thus the interior of NFP_{AB} represents all intersecting positions of *A* and *B* and the boundary represents all touching positions.

3.1.1 Calculating NFP

Slope diagram is a key aspect of an NFP so it may be useful to make a definition of slope diagram before starting calculation of an NFP.

Slope Diagram

At [12], Ghosh calls a slope diagram as extracting the relevant information by summarizing the polygon diagrammatically. The edges are represented as points on the circumference of a circle, so that the slope of a line from the center of the circle to the point is equal to the slope of the edge in question. Such a single point represents the tanget of the supporting line of the edge. These edge points are then labelled according to their counter clockwise ordering (or clockwise ordering) in the polygon as in Figure 3.1.

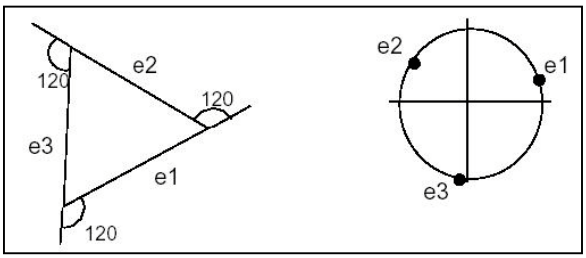

Figure 3.1 : An equilateral triangle and its slope diagram where points represent edges and arcs represent the turn of vertices.

As the edges are represented by points, the arcs between consequently labelled points represent vertices, and the angle subtended by such an arc is simply the range of tangents of the set of supporting lines that represent the vertex. Thus the slope diagram for Figure 3.1 consists of 3 points seperating 3 arcs each turning through 120°. The slope diagram can also be found as, after defining the direction of lines of a polygon, putting all lines at (0,0) point of a unit circle and marking the intersecting points of lines with this unit circle. Figure 3.2 illustrates an example of finding a slope diagram of polygon by this method [7].

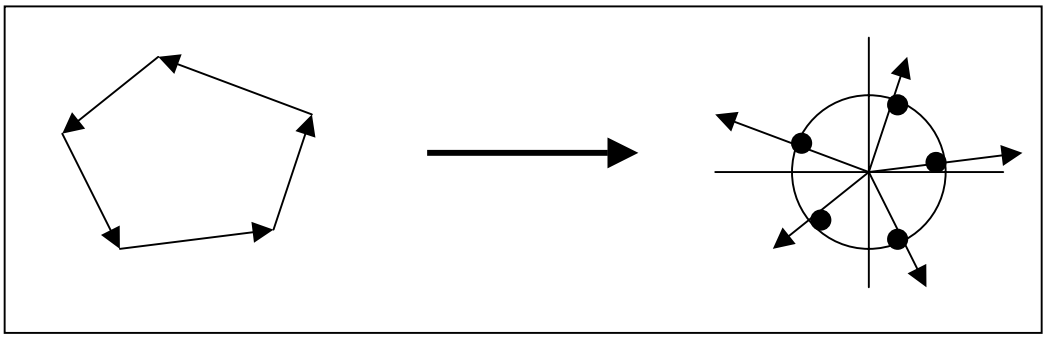

Figure 3.2 : Slope diagram of a convex polygon by defining the lines of polygon on a unit circle.

Slope diagram of a convex polygon preserves the order of edges. But if given polygon is non-convex (i.e., concave), the order of the edges at slope diagram does not preserve the order. There exists a jump to other points at the concave parts of the polygon. Figure 3.3 illustrates an example of a slope diagram for a non-convex polygon [7].

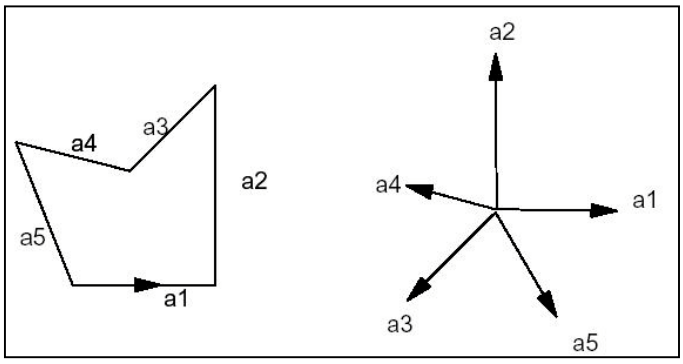

Figure 3.3 : The slope order of a polygon with concativities does not preserve the edge order.

3.1.2 Calculating NFP for Convex-Convex Polygons

Assume that the NFP represents the motion of a reference point on polygon *B* moving around polygon *A* in a counterclockwise direction. Its edges will be copies of the edges of polygon *A* oriented in a counter-clockwise direction, and copies of the edges of polygon *B* oriented in a clockwise direction. This gives an intuitive explanation of the algorithm. It starts by orienting polygon *A* counter-clockwise and polygon *B* clockwise, as shown in Figure 3.4(a). All the edges are then placed with their origin at (0,0) retaining their original direction and magnitude, as shown in Figure 3.4(b). Then, starting at an arbitrary point, the vectors are ordered according to their counter-clockwise order in the diagram and are joined end to end resulting in Figure 3.4(c). Note that the edges from each individual polygon maintain their ordering within the diagram [7,12]. This is a property of the convexity of the polygons, and ensures that each edge of the NFP is either the next edge encountered on polygon *A* or the next edge on polygon *B* as the sliding operation takes place.

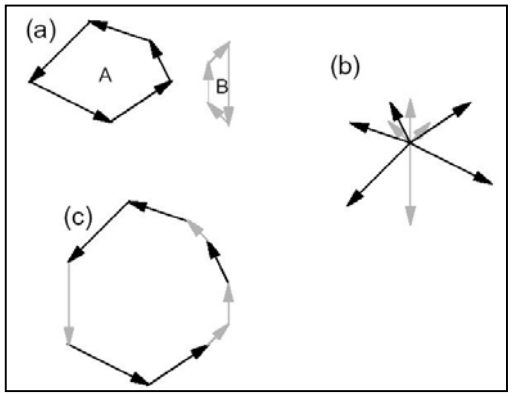

Figure 3.4 : The NFP of two convex polygons is convex and therefore can be found by sorting the edges into slope order.

3.1.3 Calculating NFP for Concave-Convex Polygons

If *A* and *B* are two arbitrary sets of points in *n*-dimensional space, then the Minkowski sum of *A* and *B* is given by [7,12]:

$$A \oplus B = \{ a + b : a \in A, b \in B\}$$

In [12], Ghosh suggested using Minkowski sums for calculating NFP for non-convex, convex polygon pairs. It is pointed out that simple vector algebra can be used to show that $A \oplus -B$, known as the Minkowski difference of A and B is equivalent to NFP_{AB}. $-B$ at the sum shows us that B is with clockwise direction where positive sets describe counter-clockwise direction.

Ghosh's method is based on the result that the Minkowski sum of any two polygons can be obtained as a function of their boundary edges. In the convex case described at previous section, the algorithm is quite simple as ordering the slopes of polygons. However for the non-convex case, some edges will appear several times in both their positive and negative orientations.

Ghosh shows that the Minkowski sum of two polygons can be obtained by merging their slope diagrams. (*Where edge points from different polygons occur in the same place we will adopt the convention of letting those from B precede those from A in the subsequent ordered list*). For the convex-convex case, this leads to exactly the same process as that given in the previous section. In the case where one polygon, say polygon A, is non-convex, the algorithm is still applicable as long as the slope diagram is traversed so that the edges of polygon A are visited in the correct order, and those edges of polygon B are included every time they are passed in the traversal. A counter-clockwise pass is interpreted as adding the edge in its given direction, and a clockwise pass as traversing the edge in the in the opposite direction.

In order to traverse the slope diagram of a non-convex polygon so that the edges are visited in the correct order, it is necessary to make several passes over some regions of the diagram, changing direction between each pass. This illustrated in Figure 3.5, where to maintain the ordering of a3 to a4, it is necessary to pass a5 and a6. To move on to a5, it is necessary to turn at a4 and move in a clockwise direction as far as a6 when another turn is necessary in order to reach a7. As all the angles between successive edge-points must be less than 180°, there is never any ambiguity as to the correct direction of travel between edge-points. Note also that the clockwise traversal represents the concavity [7].

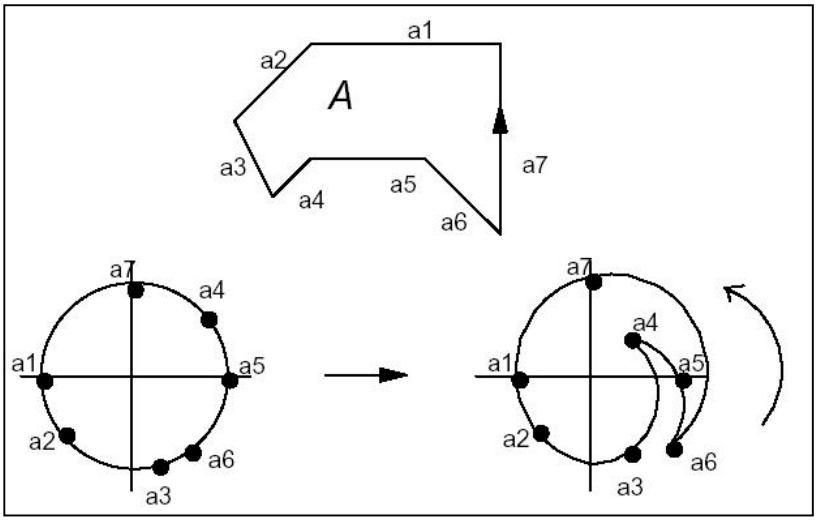

Figure 3.5 : A non-convex polygon and its slope diagram where the clockwise traversal represents concavity.

Let the polygon *A* given in Figure 3.5 be the constant polygon and, polygon *B* given in Figure 3.6 be the rotating polygon.

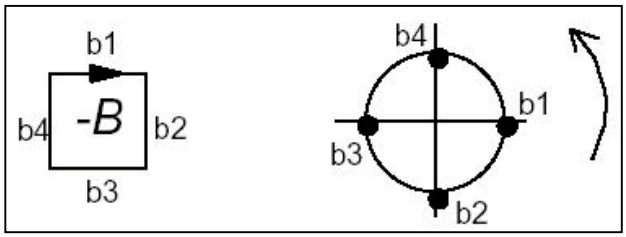

Figure 3.6 : Convex polygon and its slope diagram.

In Figure 3.7, the slope diagrams of non-convex polygon *A* and rotating polygon *B* is merged. Edge b1 is traversed 3 times, twice in the positive direction and once in the negative direction. The Minkowski difference $A \oplus -B$ is given by the sequence of edges listed as:

Merged List: b1, b4, a7, b3, a1, a2, b2, a3, b1, a4, -b1, a5, a6

NFP_{AB} will be constructed from the lines in the merged list by simply adding each line to other as vector sum [7].

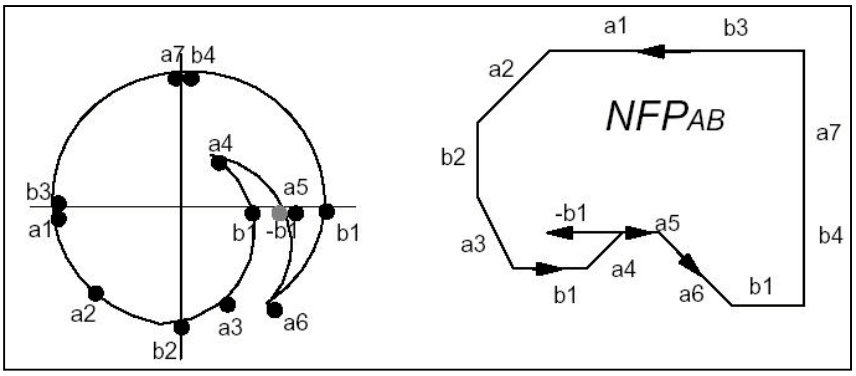

Figure 3.7 : Minkowski difference of a convex and non-convex polygon using boundary addition theorem. Merged list of addition is calculated as b1, b4, a7, b3, a1, a2, b2, a3, b1, a4, -b1, a5, a6

3.2 NFP Packing by Convex-Convex Approach

With this approach, only convex polygons are considered. Polygons are replaced one by one by a cost function $f(A,B)$. In our case, this cost function is given desired aspect ratio. After replacing the first two polygons according to cost function $f(A,B)$ by constructing NFP, a new polygon named *Addition-Polygon* is constructed from the replaced polygons which becomes the next constant polygon. Combining Addition-Polygon as constant polygon and next polygon as rotating polygon according to cost function $f(A,B)$, a new Addition-Polygon from these polygons is constructed and this iteration goes on until last polygon is replaced. After the last polygon is replaced, packing is completed.

NFP of two polygons during iteration is constructed as described in section 3.1.2. But, considering the Addition-Polygon after combining the two polygons, it can be found that Addition-Polygon does not form a convex polygon anymore. Addition-Polygon is usually a concave polygon but sometimes a non-simple polygon. Figure 3.8 illustrates the situation

with an example. Assume that Addition-Polygon of P_1 and P_2 in Figure 3.8 are replaced with the polygon in the right side of
Figure 3.8 according to cost function. The Addition-Polygon to be used as constant polygon of next iteration is therefore a non-simple polygon.

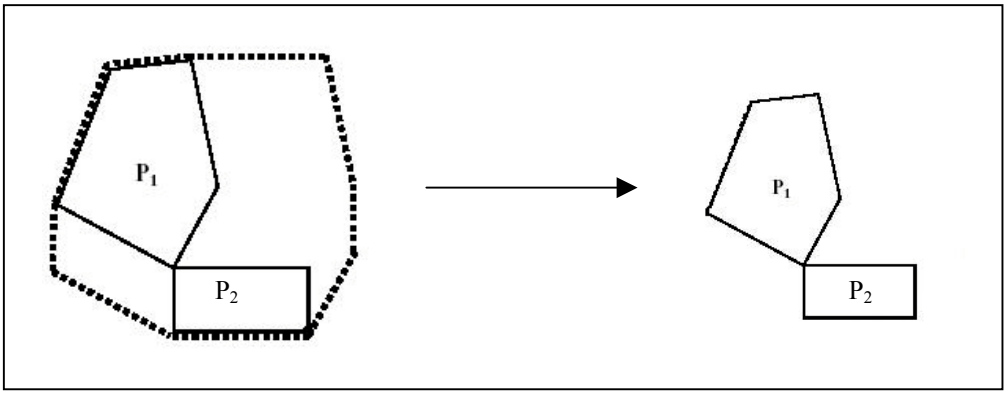

Figure 3.8 : After NFP of two polygons is calculated, replaced polygons usually form a non-simple polygon named Addition-Polygon.

However our assumption at convex-convex packing approach is that all the polygons are convex, therefore Addition-Polygon must be converted to a convex one. This is done by using a convex hull algorithm illustrated at Figure 3.9.

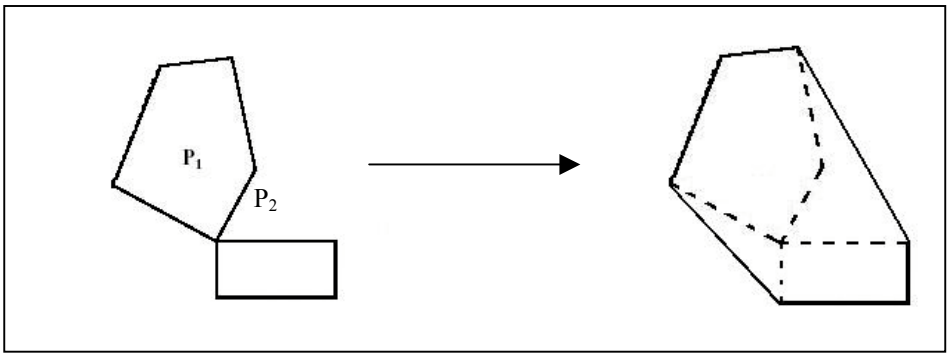

Figure 3.9 : Addition-Polygon (**left**) is convexed for next iteration. Convexed Addition-Polygon after convex hull algorithm (**right**).

After obtaining the convexed Addition-Polygon, the next polygon is taken and processed resulting in our desired graph layout with a cost function defined using aspect ratio. Figure 3.10 shows a graph layout constructed by this approach.

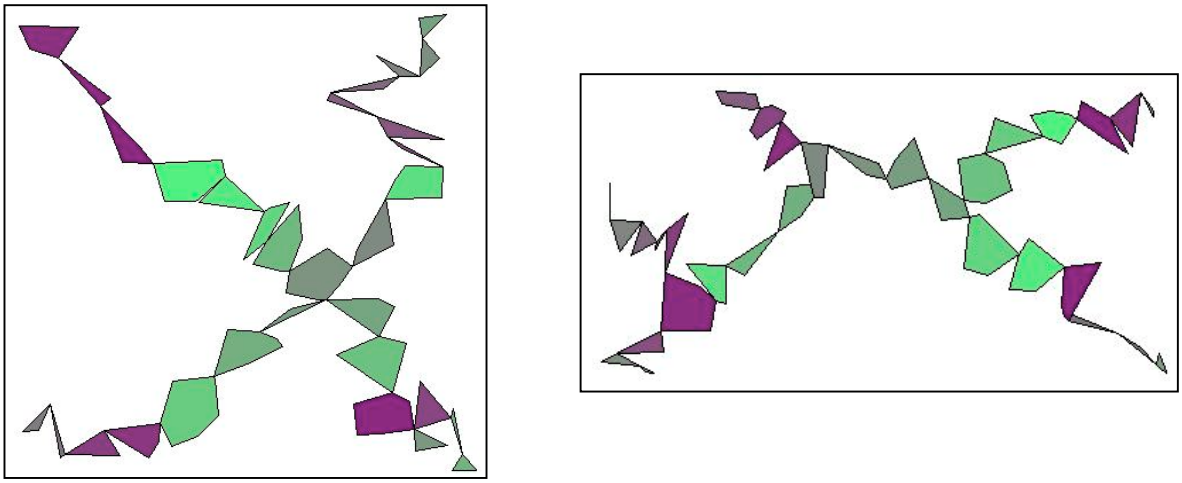

Figure 3.10 : A graph layout constructed by No-Fit Polygon Packing by Convex-Convex Approach. Graph layouts are generated for 30-40 polygons. Desired aspect ratio is 1.0 (**left**) and 2.0 (**right**)

As seen fromFigure 3., there is a large area wasted in the layout. Converging the Addition-Polygon to a convex polygon causes huge gaps between polygons. Preventing these gaps from resulting graph layout, converging Addition-Polygon to a convex polygon must be avoided. Next section describes how the converging Addition-Polygon to a convex polygon problem can be avoided to obtain a better results with NFP packing by using concave polygons for Addition-Polygon.

```
algorithm NoFitPackingConvexConvexApproach (Pi, 1 ≤ i ≤ n)
(1)   sort Pi, 1 ≤ i ≤ n in the order of nonincreasing size
(2)   foreach polygon Pi (1 ≤ i ≤ n)do
(3)         if i = 1 then
(4)               constantPolygon <- P1
(5)          else
(6)               constantPolygon <- Addition-Polygon
```

```
(7)         end if
(8)         rotatingPolygon <- P_{i+1}
(9)         merge the lines of constantPolygon and rotatingPolygon in a
   slope diagram
(10)        construct NFP from the merge list
(11)        find position of rotatingPolygon by minimizing the cost
   function.
(12)        Construct and convex the Addition-Polygon.
(13)    end foreach
```

3.3 NFP Packing by Concave-Convex Approach

In this approach, constant polygons may be concave but rotating polygons are still convex polygons. Polygons are replaced one by one by a cost function $f(A,B)$. In our case, this cost function is solely based on desired aspect ratio.

NFP of two polygons during iteration is constructed as described in section 3.1.3. But, considering the Addition-Polygon after combining the two polygons, it can be founded that Addition-Polygon may not form a simple polygon anymore. Addition-Polygon becomes at least a concave polygon but usually a non-simple polygon after the combination. Figure 3.11 illustrates the situation briefly.

The assumption of this approach is that constant polygons must be at least concave polygons. But, as seen in Figure 3.11, the Addition-Polygon may be a non-simple polygon, therefore we have to convert the non-simple polygon to a simple concave polygon or to a simple convex polygon. To do this, a `Concave-Connector` method is designed and applied to Addition-polygons.

After obtaining the Addition-Polygon by using `Concave-Connector` method, iteration on next polygon until last polygon results our desired graph layout with cost function based on aspect ratio. Figure 3.11 shows a graph layout constructed by this approach.

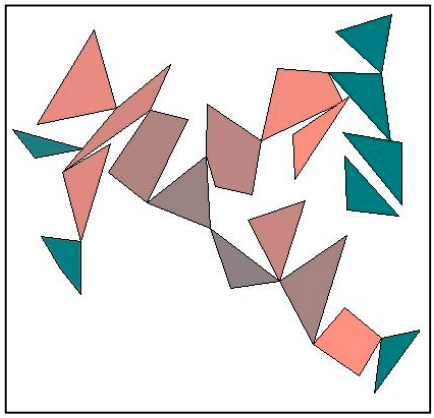

Figure 3.11 : A graph layout constructed by No-Fit Polygon Packing by Concave-Convex Approach.

As seen from Figure 3.11, the graph layout is better than Convex-Convex approach. Constructing Addition-Polygon by using `Concave-Connector` algorithm saves considerably wasting area.

```
algorithm NoFitPackingConcaveConvexApproach (Pi, 1 ≤ i ≤ n)
(1)   sort Pi, 1 ≤ i ≤ n in the order of nonincreasing size
(2)   foreach polygon Pi (1 ≤ i ≤ n) do
(3)         if i = 1 then
(4)               constantPolygon <- P1
(5)          else
(6)               constantPolygon <- Addition-Polygon
(7)         end if
(8)         rotatingPolygon <- Pi+1
(9)         merge the lines of constantPolygon and rotatingPolygon in a
   slope diagram by using Concave-Convex approach.
(10)        construct NFP from the merge list
(11)        find position of rotatingPolygon by minimizing the cost
   function.
(12)        Addition-Polygon <- Concave-Connector(ConstantPolygon,
   rotatingPolygon)
(13)   end foreach
```

3.3.1 Concave-Connector Method

This algorithm connects two polygons and forms a concave or convex Addition-Polygon. It is a recursive algorithm not taking into account the following two specified cases: These simple cases occur when there exist only one intersection point:

a) The intersection point may be a vertex of each polygon (vertex-vertex intersection),
b) The intersection point may be a vertex on one of the polygons and this vertex lies on a line of other polygon (vertex-line intersection).

Figure 3.12 illustrates these two cases with examples.

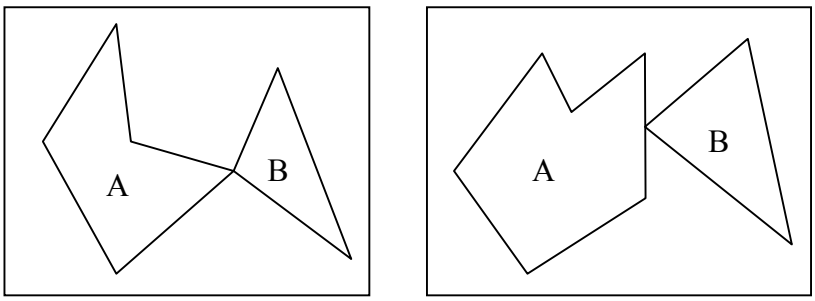

Figure 3.12 : Two cases of one intersection point. A vertex-vertex intersection (**left**) and vertex-line intersection (**right**).

Below is the pseudocode for the algorithm to Concave-connector:

```
algorithm ConcaveConnector Constant = {C1,…, Cs}, Rotating = {R1,…, Rt}
(1)   Find intersection vertice(s).
(2)   if a vertex of Constant intersects a vertex of Rotating then
(3)         VertexVertexIntersection(Constant, Rotating)
(4)   else if a line of Constant intersects a vertex of Rotating OR a
      vertex of Constant intersects a line of Rotating then
(5)         LineVertexIntersection(Constant, Rotating)
(6)   else
(7)         ComplexIntersection(Constant, Rotating)
(8)   end if
```

Vertex-Vertex Intersection Method

NFP calculation assumes that the rotation of the constant polygon is in counter-clockwise direction, and the rotating polygon moves in clockwise direction. We mark each points of the polygons according to their direction. We add label 'C' before the vertex for the constant polygon, and label 'R' before the vertex for the rotating polygon. Now our polygons are labelled as *ConstantPolygon* = $\{C_1,\ldots, C_s\}$, and *RotatingPolygon* = $\{R_1,\ldots, R_t\}$ where s is the number of vertices of constant polygon and t is the size of rotating polygon. Assume k^{th} vertex of constant polygon and l^{th} vertex of rotating polygon is the intersecting vertex. Then, since orientations of each polygon is different, $\{C_{k-1}, C_k = R_l, R_{l-1}\}$ and $\{C_{k+1}, C_k = R_l, R_{l+1}\}$ form triangles as illustrated in Figure 3.13.

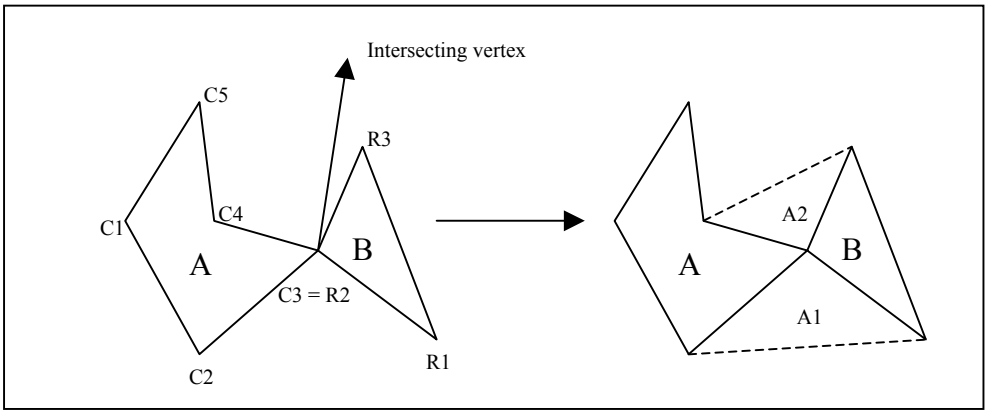

Figure 3.13 : The triangles of vertex-vertex intersection. A1 = $\{C_2, C_3 = R_2, R_1\}$ and A2 = $\{C_4, C_3 = R_2, R_3\}$

After finding the triangles A1 and A2, we calculate the area of these triangles and connect the polygons from the vertices of the minimum area triangle that is new Addition-Polygon becomes either *Addition-Polygon* = $\{C_1, C_2, \ldots, C_k = R_l, R_{l-1}, \ldots, R_1, R_t, \ldots, R_{l+1}, C_{k+1}, \ldots, C_s\}$, or *Addition-Polygon* = $\{C_1, C_2, \ldots, C_k = R_l, R_{l+1}, \ldots, R_t, R_1, \ldots, R_{l-1}, C_{k+1}, \ldots, C_s\}$. Assume that the given example at Figure 23 has the minimum area triangle as A1. Then the Addition-Polygon becomes *Addition-Polygon* = $\{C_1, C_2, C_3 = R_2, R_1, R_3, C_4, C_5\}$ as illustrated in Figure 3.14.

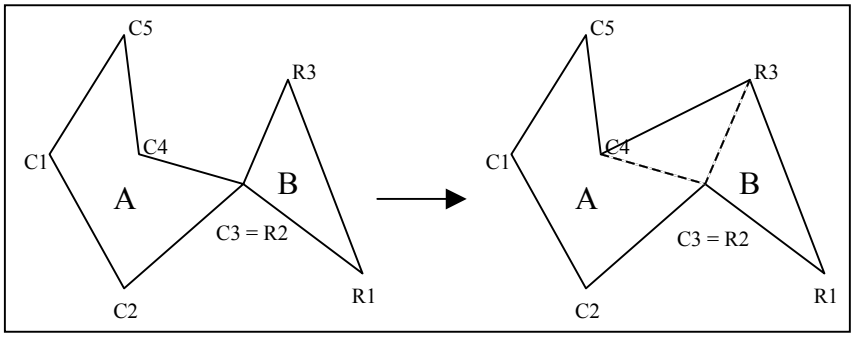

Figure 3.14 : The Addition-Polygon of two intersected polygons (**right**) becomes a simple polygon (**left**).

Below is the pseudocode for the algorithm to Vertex-Vertex intersection:

algorithm VertexVertexIntersection *Constant* = {C_1,…,C_s},*Rotating* = {R_1,…,R_t}

(1) Label all the points of each polygon according to their orientation.

(2) Find the intersection point.

(3) Get the two triangles around the intersection point formed by {C_{k-1}, C_k = R_l, R_{l-1}} and {C_{k+1}, C_k = R_l, R_{l+1}} where C_k = R_l is the intersection point.

(4) Calculate the area of each triangle.

(5) Connect the vertices according the minimum area triangle as forming *Addition-Polygon* = {C_1, C_2, …, C_k = R_l, R_{l-1}, …, R_1, R_t, R_{t-1},…, R_{l+1}, C_{k+1}, …, C_s}, or *Addition-Polygon* = {C_1, C_2, …, C_k = R_l, R_{l+1}, …, R_t, R_1, …, R_{l-1}, C_{k+1}, …, C_s}

Vertex-Line Intersection Method

This case is an extension of the vertex-vertex algorithm. If the intersection point is on the line of a polygon, then adding this intersection point as a vertex to the polygon which is intersected on line converts this case to the vertex-vertex algorithm case described earlier. Figure 3.15 illustrates this situation. Then, applying the same steps applied at vertex-vertex intersection algorithm gives the Addition-Polygon.

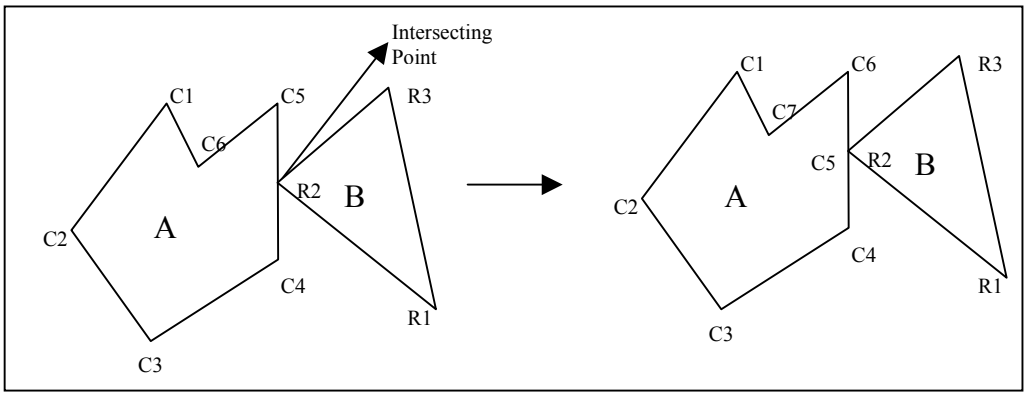

Figure 3.15 : Adding imtersection point {C5} to the polygon *A* as a new vertex converts the case to the case described earlier.

Below is the pseudocode for the algorithm to Vertex-Line intersection:

```
algorithm VertexLineIntersection Constant = {C1,…,Cs}, Rotating = {R1,…,Rt}
(1)   Label all the points of each polygon according to their orientation.
(2)   Add intersecting point as a new vertex to the polygon where
   intersecting point is on a line of this polygon and relabel.
(3)   VertexVertexIntersection(Constant, Rotating).
```

Complex Intersection Method

If the Addition-Polygon does not fit one of the cases described above, then we apply an algorithm called *Comlex Intersection*. This case occurs when more than one intersection occurs, or when line intersections occur. Figure 3.16 illustrates two examples of such kind of Addition-Polygons. In this algorithm, all vertices are labelled from 1 to *s*+*t* where *s* is the number of vertices of constant polygon, and *t* is the number of vertices of rotating polygon.

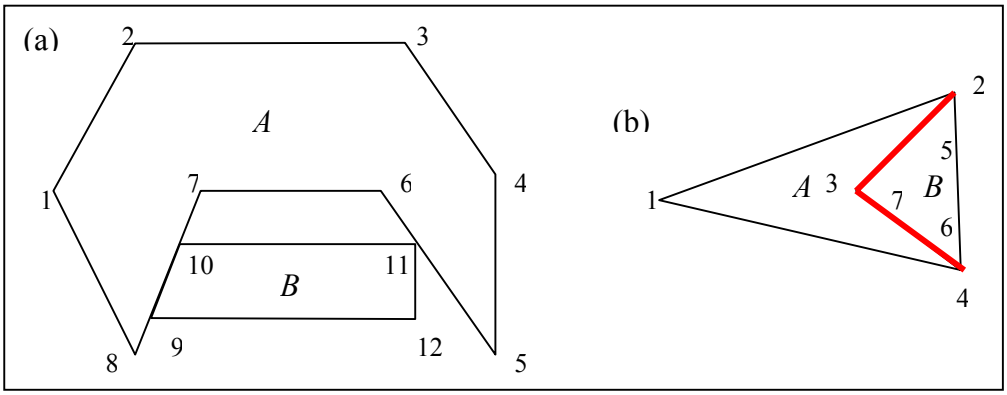

Figure 3.16 : Two examples for Complex Intersection case: two vertex intersection (**left**), and two line intersection at (**right**).

The algorithm is a recursive one that finds all the paths from a non-complex vertex to itself. Non-complex vertex is a vertex of constant polygon that does not intersect with a line or a vertex of rotating point. Polygon *A* of Figure 3.16(a) has all vertices as non-complex vertex but Poygon *B* of Figure 3.16(b) has only vertex 1 as non-complex vertex. If a vertex of constant polygon and a vertex of rotating polygon intersects, then connecting the labels of this vertex and considering it as one vertex will solve the problem. In Figure 3.16(b), vertex 2 of constant polygon and vertex 5 of rotating polygon intersects so considering this vertex as vertex (2,5) will solve this problem.

Also, finding and adding the vertices that are on any line of the other polygon (if vertex is element of rotating polygon, other polygon is constant polygon, or if vertex is element of constant polygon, other polygon is rotating polygon) to this polygon, will result as finding all the consequent vertecies of this vertex. In Figure 3.16(a), if vertex 11 is not added to constant polygon, then its consequent vertices will only be vertex 10 and 12. If we add this vertex to constant polygon, then consequent vertices will be vertices 5, 6, 10 and 12.

Another important definition of algorithm is *First Branching Vertex* (FBV). This means the vertex on the path where the first branching occurs. In Figure 3.16(b), FBV is vertex (2,5).

The algorithm starts from a non-complex vertex and finds its consequent vertex or vertices. The direction can not return back to its previous vertex. If FBV is met, then starts branching

the path. All the paths is found after branching. If any path after FBV meets FBV (itself) on its way, this branch is killed. Remaining paths include a path of starting polygon so kill this path too. If there exists more than one path at remaining paths, kill paths that does not include all non-cutting vertices. If there still exists more than one path, construct the polygons from the paths, calculate the areas and the maximum area path is the path of Addition-Polygon.

Figure 3.17 illustrates an example of finding Addition-Polygon of Figure 3.16(a) and Figure 3.19 illustrates an example of finding Addition-Polygon of Figure 3.16(b).

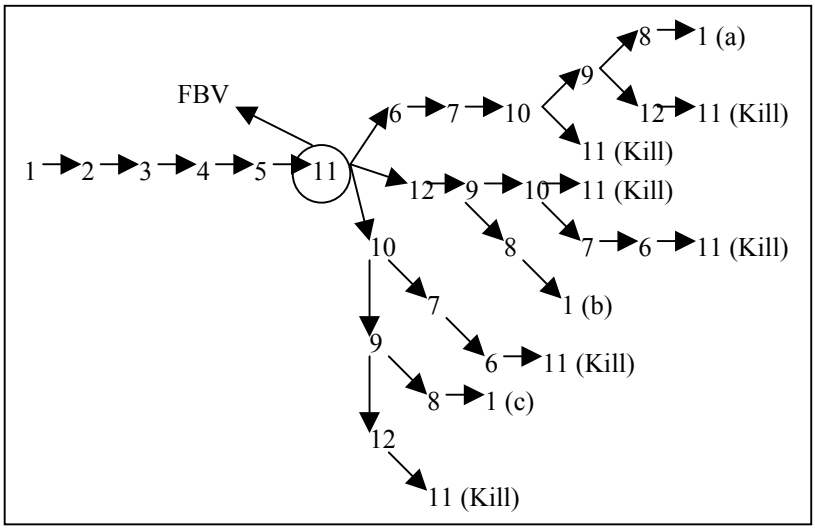

Figure 3.17 : The application of algorithm to the Addition-Polygon in Figure 26(a).

Therefore, we obtain three paths as result of algorithm. These paths are:

(a) {1,2,3,4,5,11,6,7,10,9,8,1}
(b) {1,2,3,4,5,11,12,9,8,1}
(c) {1,2,3,4,5,11,10,9,8,1}

Path (a) is the starting polygons itself, so it must be killed. There is only one non-cutting vertex of second polygon, vertex 12, and path (c) does not include this vertex so kill it too. Then, only path (b) remains, and it is the path of Addition-Polygon as seen at Figure 3.18.

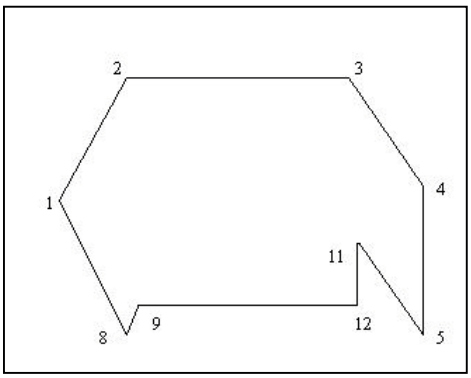

Figure 3.18 : Addition-Polygon of polygons in Figure 3.16(a)

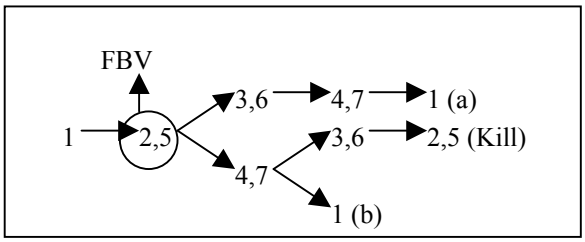

Figure 3.19 : The application of algorithm to the Addition-Polygon in Figure 3.16(b).

Therefore, we obtain two paths as result of algorithm. These paths are:

(a) {1,(2,5),(3,6),(4,7),1}
(b) {1,(2,5),(4,7),1}
Path (a) is the starting polygons itself, so it must be killed. Then, only path (b) remains, and it is the path of Addition-Polygon as seen in Figure 3.20.

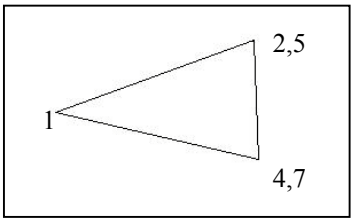

Figure 3.20 : Addition-Polygon of polygons in Figure 3.16(b)

Below is the pseudocode for the algorithm to Complex intersection:

```
algorithm ComplexIntersection Constant = {C1,…, Cs}, Rotating = {R1,…, Rt}
(1)    Find intersection points.
(2)    Combine vertices if they intersects.
(3)   Add vertices to the appropriate polygon if a vertex is on the line of
   this polygon.
(4)   Find a non-complex vertex as startingVertex.
(5)   Define a set of paths as Paths
(6)   Paths <- FindPath (startingVertex).
(7)   foreach path in Paths do
(8)         if path is starting polygon path then
(9)               remove this path from paths
(10)        else if path does not have any of the non-cutting vertices then
(11)  remove this path from paths
(12)        end if
(13)  end foreach
(14)  if there exists more than one path in Paths then
(15)        Construct the polygons
(16)        return the path of the polygon that has maximum area
(17)  else
(18)        return the path
(19)  end if

algorithm FindPath (Vertex startingVertex)
(1)   Get next vertex of startingVertex as nextVertex.
(2)   if nextVertex is branching then
(3)         if FBV is null then
(4)               FBV <- nextVertex
(5)               FindPath(nextVertex)
(6)         else if nextVertex = FVB then
(7)               return
(8)         else if nextVertex = startingVertex then
(9)               return path
(10)        else
(11)              FindPath(nextVertex)
(12)        end if
(13)  else
(14)        add nextVertex to path
```

```
(15)  end if
```

3.4 Time Complexity Analysis of NFP Approaches

Let the number of graph objects given as polygons be n and the average number of corners per object be m.

In No-Fit Polygon Packing Convex-Convex approach, we pass over each polygon and merge slope diagrams of these polygos. To merge the created slope diagrams, we pass for each corner of two objects, one is the no fit polygon and the other is the iterated polygon. Also, we re-calculate all the positions of polygons. Therefore, the complexity of algorithm becomes $O(n \cdot m^2 \cdot n + (n+m) \cdot m \cdot n)$ which can be stated as $O(n^2 \cdot m^2)$, which is quadratic in the number of objects.

In No-Fit Polygon Packing Concave-Convex approach, we pass over each polygon and create NFP of these polygons. To create NFP, we pass for each corner constant object and may pass for the rotating polygons corners $c \cdot m$, where c is the constant number of concavities of the constant polygon. Note that $c < m$. Also, we re-calculate all the positions of polygons. The Concave-Connector method has complexity $O(m)$ and is calculated for each pass of polygons. Therefore, the complexity of algorithm becomes $O(n \cdot c \cdot m^2 \cdot n \cdot m + (c \cdot m^2) \cdot m)$ which can be stated as $O(n^2 \cdot m^3)$, which is again quadratic in the number of disconnected objects to be laid out.

4. Implementation and Discussion

In this chapter, implementation details and results will be given.

4.1 Implementation Platform and Environment

Java Development Kit 1.3.1 is used for implementation.

Implementation is done on a machine that has the following hardware devices:

- AMD Athlon XP 1700+,
- 256 MB DDR Ram,
- 40 GB HD,
- 32 Mb Nvidia Riva TNT2 Model 64 Graphic Adapter.

Minimum requirements to run the implementation may be given as:

- Pentium II 300 MHz or compatible processors,
- 32 MB SD Ram,
- 1 GB HD free space,
- 1 MB graphic adapter.

4.2 Implementation Interface

Implementation consists of two user interfaces. A main window where the user enters the required parameters such as aspect ratio, minimum and maximum number of objects to be created, the maximum number of object corners, etc. The second window consists of the output screen of resulting layout. User can return to main interface and continue to generate objects and their graph layouts. For displaying the polygons, all the polygons are repositioned and scaled to fit into the frame [14] . Figure 4.1 describes these interfaces.

Figure 4.1 : The user interfaces: main windo (**left**) and Layout window (**right**).

The description of the components of the Main window is as follows:

- Enter Min. and Max number of objects:

`Left TextBox` : Minimum number of objects to be created.

`Right TextBox` : Maximum number of objects to be created.

- Enter Max. width and height of each object:

`Left TextBox` : Maximum width of objects to be created.

`Right TextBox` : Maximum height of objects to be created.

- Enter desired aspect ratio: The desired aspect ratio.
- Enter max. no. of corners per object: The maximum number of corners of objects to be created

- Enter no. of runs: How many number of runs to be done for obtaining test results
- "`Create Objects`" Button: Creates the objects according to given parameters above.
- "`Start Batch`" Button: Runs the application for given number of runs for test results.
- "`Quit`" Button: Quits the application.

After objects are created, the following buttons are displayed for obtaining graph layout for created disconnected objects:

- "`Apply Tiling`" Button: Applies tiling packing approach to created objects.
- "`Apply Alternate-Bisection Button`": Applies alternate-bisection packing approach to created objects.
- "`Apply Polyomino Packing`" Button: Applies polyomino packing approach to created objects.
- "`Apply No Fit Packing`" Button: Applies no fit packing approach to created objects.
- "`Apply Convex No Fit Pack…`" Button: Applies no fit packing convex-convex approach to created objects.

The description of the components of the Layout window inteface is as follows:

- "`Show/Hide Grid`" Button: For the polyomino packing approach, it toggles showing or hiding the grid of polyominos.
- "`Return Main`" Button: Returns the main interface.
- "`Fill/No Fill Objects`" Button: Fills or removes the fills inside the graph objects.
- "`Quit`" Button: Quits the application.

4.3 Implementation Results

The *adjusted fullness* of a packing expresses the fillness of packing, in percentage, with respect to the desired aspect ratio. To be precise, it considers the additional area wasted when the final drawing is displayed in a region of desired aspect ratio.

We have performed a number of executions to compare tiling, alternate-bisection, polyomino, NFP Convex-Convex approach and NFP Concave-Convex approach packings. Graphs that contained up to thirty disconnected objects were used where each object was assumed to be a polygon with random number of corners in {5,10}, each with random integer coordinates in [1…100], all independent and uniformly distributed. For the tiling and alternate-bisection methods the tightest rectangles bounding these polygons were used, for polyomino packing the smallest polyomino bounding the polygons were used. For each parameter change, the results were calculated for 5 times and average of the time and adjusted fullness is taken. Same graph objects are used for each set of the experiment. The performance comparison is presented in Figure 4.2 through 4.5

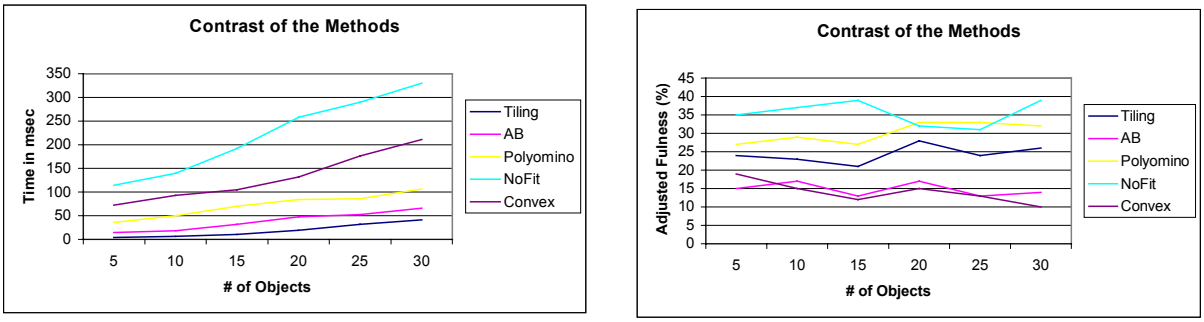

Figure 4.2 : Comparison of packing methods with Desired Aspect Ratio 1.0, and maximum number of corners per object is 5. Time in miliseconds versus number of objects (**left**) and Adjusted Fullness percentage versus number of objects (**right**).

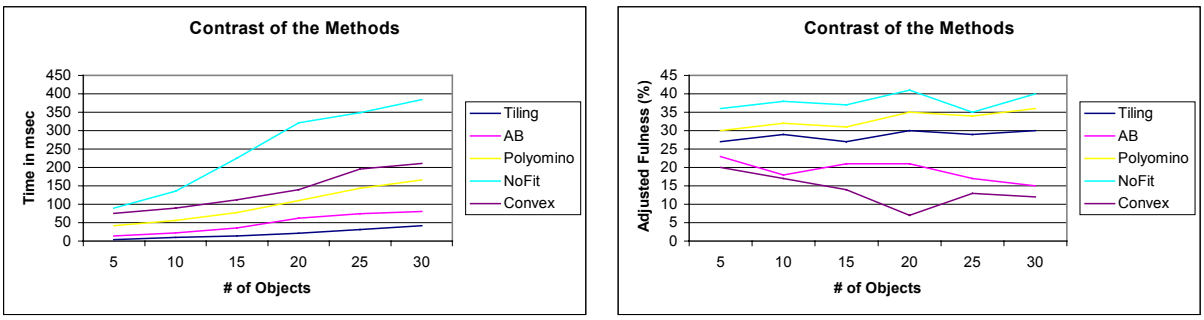

Figure 4.3 : Comparison of packing methods with Desired Aspect Ratio 1.0, and maximum number of corners per object is 10. Time in miliseconds versus number of objects (**left**) and Adjusted Fullness percentage versus number of objects (**right**).

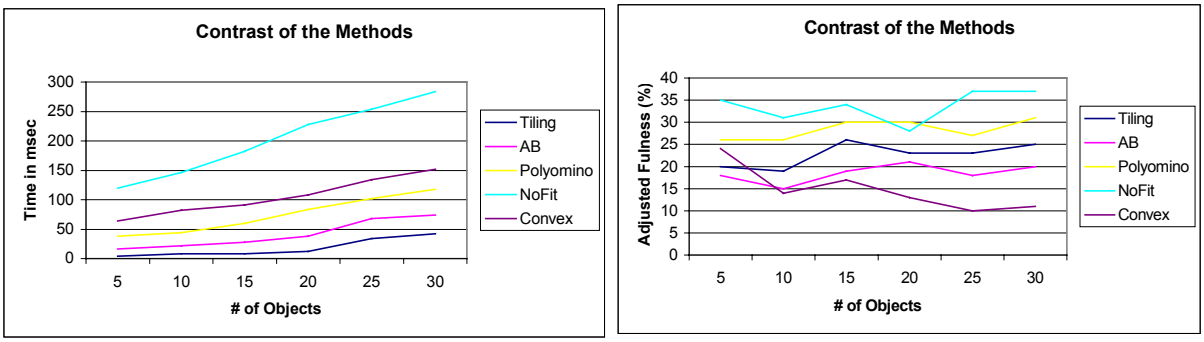

Figure 4.4 : Comparison of packing methods with Desired Aspect Ratio 1.5, and maximum number of corners per object is 5. Time in miliseconds versus number of objects (**left**) and Adjusted Fullness percentage versus number of objects (**right**).

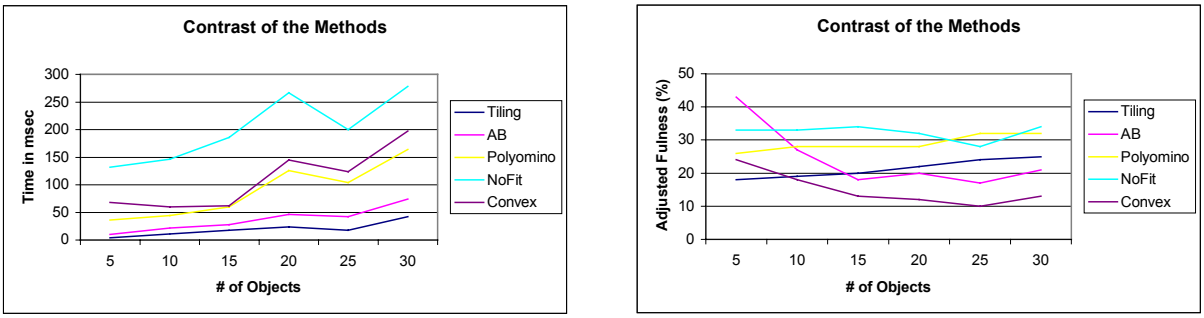

Figure 4.5 : Comparison of packing methods with Desired Aspect Ratio 2.0, and maximum number of corners per object is 5. Time in miliseconds versus number of objects (**left**) and Adjusted Fullness percentage versus number of objects (**right**).

The following figures contrast NFP packing Concave-Convex approach by Polyomino packing when the grid size is half of the size (Figure 4.6) of the one in Figure 4.2 and when the grid size is double the size (Figure 4.7) of the one in Figure 4.2. As it is clear from the figures, the NFP Concave-Convex approachs perform better in terms of the quality of the layout even when the grid is quite fine.

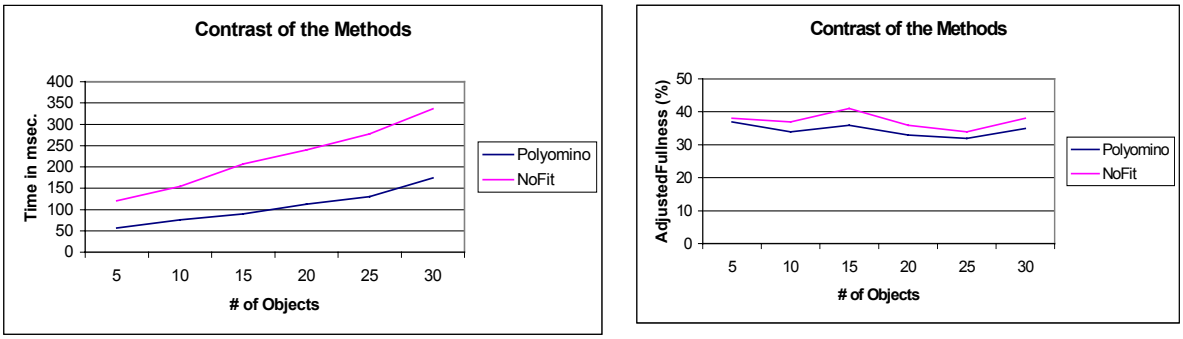

Figure 4.6 : Comparison of Polyomino packing with NFP Packing Concave-Convex approach. Desired Aspect Ratio 1.0, and maximum number of corners per object is 5, grid size of the Polyominoes is 5, half of the value of in Figure 4.2. Time in miliseconds versus number of objects (**left**) and Adjusted Fullness percentage versus number of objects (**right**).

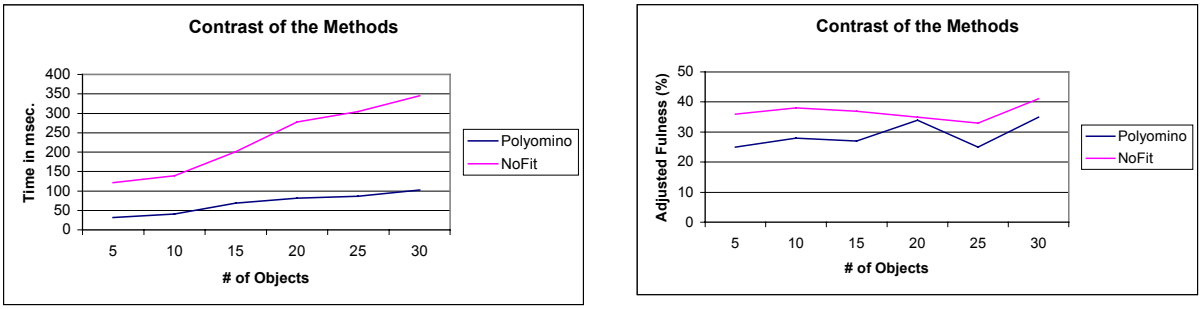

Figure 4.7 : Comparison of Polyomino packing with NFP Packing Concave-Convex approach. Desired Aspect Ratio 1.0, and maximum number of corners per object is 5, grid size of the Polyominoes is 20, double of the value of in Figure 4.2. Time in miliseconds versus number of objects (**left**) and Adjusted Fullness percentage versus number of objects (**right**).

4.4 Discussion

As seen from in Figure 4.2, through 4.5, No-Fit Polygon packing Concave-Convex approach results in the most compact drawings, about more than 35% adjusted fullness. The best graph layout after our new approach is Polyomino packing approach. No-Fit Polygon packing Concave-Convex approach sometimes (≈5%) draw worst graph layout than polyomino

packing but these cases rises from the fact that most of the polygon shapes in sorted order does not completment each other.

The time complexity of No-Fit Polygon packing Concave-Convex approach is more than other algorithms so it uses more execution time, it is slower than other algorithms. But this execution time is still in acceptable bounds for practical purposes

No-Fit Polygon packing Convex-Convex approach gives almost the worst adjusted fullness. Since this approach converts the Addition-Polygon to a convex polygon, lots of space is waste. Sometimes, the adjusted fullness is lower than the tiling algorithm. In addition to this fact, it has the second highest execution time after No-Fit Polygon packing Concave-Convex approach.

Increasing the number of corners per object rises the execution time of No-Fit Polygon packing Concave-Convex approach, but changing aspect ratio, does not seem to affect the execution time since it always tries all the possible places of each polygon.

Decreasing the grid size of the Polyomino packing, increses the adjusted fullness of graph layout but costs to a higher execution time. Increasing the grid size of Polyomino packing, decrases the adjusted fullness of graph layout but the execution time is much better in this case. This comparison can be seen by Figure 4.6 and 4.7 according to Figure 4.2. In both experiments, it is observed that NFP Packing Concave-Convex approach still have better adjusted fullness on same set of objects againist Polyomino Packing, but still costs to a higher execution time.

Therefore, No-Fit Polygon packing Concave-Convex approach can be used for two-dimensional packing problems as it results in best graph layouts with acceptable execution time.

5. Conclusion and Future Work

In this thesis, we have aimed to show that No-Fit Polygon Packing can be used for two dimensional packing. This approach is compared with existing packing algorithms. Comparison is done through adjusted fullness and time analysis. We have developed and implemented all the algorithms described in the thesis. For implementing No-Fit Packing, several methods including Concave-Connector method is designed and implemented as helper methods.

The contribution of this thesis was the design of a new approach and the analysis to contrast it with tiling, alternate-bisection and polyomino packings. The previous algorithms approximate polygons with rectangles or polyominoes whereas the new No-fit polygon packing approach calculates the graph layout directly with given polygons. Expected result from this contrast was No-Fit Polygon packing shall have better graph-layout performance, but worse time-complexity against other approaches. The experiment results verify these predictions.

Calculating NFP of two polygons is a time consuming operation, hence improving this calculation may decrease the execution time of the algorithm. For future work of No-Fit Polygon Concave-Convex approach, optimizing the implementation may reduce execution time.

After the resulting graph layouts are examined, we found that using No-Fit Polygon Concave-Concave approach described in [7] can increase the adjusted fullness of graph layout. In this method, there will be no restrictions if given graph objects are convex polygons. Also, by removing the sorting operation before packing the initial graph objects, we can obtain better graph layouts whereas this causes slower execution time.

To conclude, No-Fit Polygon packing Concave-Convex approach can be used for disconnected graph layout resulting in more compact drawings with slower but acceptable execution times.

BIBLIOGRAPHY

[1] Freivalds, K., Dogrusoz, U., and Kikusts. P., Disconnected Graph Layout and The Polyomino Packing Approach, *Proceedings of Graph Drawing 2001, Lecture Notes in Computer Science*, vol. 2265, 378-391, 2002.

[2] Dogrusoz, U., Two-Dimensional Packing Algorithms For Layout of Disconnected Graphs, *Information Sciences*, vol. 143/1-4, 147-158, 2002.

[3] Li, Z., Compaction Algorithms For Non-Convex Polygons, *PhD thesis, Department of Computer Sciences, University of Harvard, Cambridge, Massachusetts,* May 1994.

[4] Burke, E., and Kendall, G., Evaluation of Two Dimensional Bin Packing Using The No Fit Polygon, *Computers and Industrial Engineering,* vol. 26, December 1999.

[5] Burke, E., and Kendall, G., Implementation and Performance Improvement of the Evaluation of Two-Dimensional Bin Packing Problem Using The No Fit Polygon, *University of Nottingham, Technical Report # ASAP99001*, 1999.

[6] Burke, E., and Kendall, G., Applying Ant Algorithms and the No Fit Polygon to the Nesting Problem, *Lecture Notes in Artificial Intelligence*, 1747, 453-464, December 1999

[7] Bennel, J.A., Dowsland, K.A., and Dowsland, W.B., The Irregular Cutting Stock Problem-A New Procedure For Deriving The No-Fit Polygon, *Computers and Operations Research*, vol. 28, 271-287, 2001.

[8] Coffman, G.G., Garey, M.R., Johnson, D.S., and Tarjan R.E, Performance Bounds For Level-Oriented Two-Dimensional Packing Algorithms, *Society for Industrial and Applied Mathematics,* vol.9: No 5*, 808-826,* November 1980.

[9] Baker, B.S., Coffman, E.G., JR and Rivest, R.L., Orthogonal Packings in two Dimensions, *Society for Industrial and Applied Mathematics*, vol. 9, no.4, pp. 846-855, November 1980.

[10] Baker, B.S., Brown, D.J., and Katseff, H.P., A 5/4 Algorithm for Two-Dimensional Packing, *Journal of Algorithms*, vol. 2, 348-368, 1981.

[11] Adamowicz, M., and Albano, A., Nesting Two Dimensional Shapes In Rectangular Modules, *Computer Aided Design*, vol. 8, No. 1: 27-33, 1976.

[12] Ghosh, P. K., A Unified Computational Framework For Minkowski Operations, *Computers and Graphics*, 17, 4: 357-378, 1993.

[13] Albano, A., and Sapuppo, G., Optimal Allocation of Two-Dimensional Irregular Shapes Using Heuristic Search Methods, *IEEE Transactions on Systems Man. And Cybernetics*, SMC-10, 5, 242-248, May 1980.

[14] U. Dogrusoz, Q. Feng, B. Madden, M. Doorley, and A. Frick, Graph Visualization Toolkits, *IEEE Computer Graphics and Applications*, vol. 22, No. 1: 30-37, January/February 2002.