



**Linnaeus University**

School of Computer Science, Physics and Mathematics

Degree project

# A Space-Filling Technique for the Visualization of Planar st- Graphs



*Author:* Yuanmao Wang  
*Date:* 2012-08-07  
*Subject:* Computer Science  
*Level:* Bachelor  
*Course code:* 2DV00E

## Abstract

Graphs currently attract an increasing number of computer scientists due to their widely adoptions in different areas. However, when people perform graph drawing, one of the most critical issues they need to concern is aesthetics, i.e., to make the graph more suitable for human perceptions. In this work, we will aim at exploring one specific kind of graph "*planar st-graphs*" with space-filling technique in Info Vis area. We would cover *edge crossing elimination, layer assignment, graph drawing algorithms, and new development of space-filling technique in planar st-graphs drawing etc.* The final aim of this project is to develop a new algorithm to draw planar st-graphs based on a space-filling visualization approach with minimum edge crossings and maximum space usage.

**keywords:** Planar st-graph, Space-filling, Information Visualization, Edge crossing, Graph Drawing.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	Problems . . . . .	2
1.3	Motivations . . . . .	4
1.4	Project Goals . . . . .	5
1.5	Restrictions . . . . .	5
1.6	Structure of the report . . . . .	6
<b>2</b>	<b>Related Works</b>	<b>7</b>
2.1	Information Visualization . . . . .	7
2.2	Planar st-graph . . . . .	7
2.2.1	St-graph . . . . .	8
2.2.2	Planar Graph . . . . .	8
2.3	Space-Filling . . . . .	10
2.3.1	Treemap . . . . .	10
2.3.2	Radial Space-Filling . . . . .	11
2.4	Parameterized st-Orientations . . . . .	12
2.5	Processing . . . . .	12
<b>3</b>	<b>Visualization</b>	<b>14</b>
3.1	Data Source . . . . .	14
3.2	Initial View . . . . .	15
3.3	Layer Assignment and X Coordinate Assignment . . . . .	15
3.3.1	Layer Assignment . . . . .	16
3.3.2	X-coordinate Assignment . . . . .	18
3.3.3	Creation of Dummy Nodes . . . . .	19
3.4	Eliminate Edge Crossings Manually . . . . .	20
3.5	Drawing Arcs . . . . .	21
3.6	Interactions . . . . .	22
<b>4</b>	<b>Implementation</b>	<b>23</b>
4.1	Original Planar St-Graph Visualization . . . . .	23
4.1.1	Layer and X-coordinate Assignment . . . . .	23
4.1.2	Generating Dummy Node . . . . .	27
4.1.3	Edge Crossing Reduction . . . . .	27
4.1.4	Original Graph Layout . . . . .	28
4.2	Space-filling Visualization . . . . .	28
<b>5</b>	<b>Discussion</b>	<b>33</b>
<b>6</b>	<b>Future Work</b>	<b>35</b>

# 1 Introduction

Comparing with a table or a paper with thousands of words, people often prefer a diagram or picture instead to make a much better sense of what they want to know. Indeed, graphs can convey information into a much attractive and easy way, especially when an organization need to know the work flow diagram, then usually a kind of flow chart will be conducted in order to make it clear for all people to recognize the approaching events and possible and related activities.

Comparing with diagrams and images, while in computer science or mathematics, there is one concept called "graph" which as Diestel, R. [1] defined "*A graph is an abstract representation of a set of objects where some pairs of the objects are connected by links*". The concept of graph will be introduced in the coming Section 1.1. After that, we will explain the idea about how graphs are used in our lives and in computer science (e.g. Graph Drawing) where we will also introduce the concept of planar st-graphs which is the main target of this study. Further, we will talk about the problems with planar st-graphs and show the goal of the project.

## 1.1 Background

What is a graph? Actually, when we talk about graph, it mainly refer to the object in "*Graph Theory*" which is a branch of Mathematics. The research in "*Graph Theory*" is based on "*Graph*" which consists with several given nodes and links connect them. This kind of graphics usually used for describing the specific relations between objects where the nodes represent objects and the links represent relations [1]. In the most common sense of the definition of graph, it is an ordered pair  $G = (V, E)$  comprising a set  $V$  of vertices or nodes together with a set  $E$  of edges or lines. For undirected graphs, it is a graph  $A$ , in which edges have no orientation, i.e., they are not ordered pairs, but sets  $u, v$  of vertices [2]. While a directed graph is an ordered pair  $D = (V, A)$  with  $V$  a set whose elements are called vertices, and  $A$  is a set of ordered pairs of vertices, called directed edges. Comparing with these two kinds of graphs, the main difference is ones vertices are ordered and the other is not. I.e., the edges have an orientation.

Today, with the development of technology, increasing works can be done by computers. One of the mostly widely used techniques is computer graph. As Michael T.G. & Stephen G.K. [2] stated "*Graph drawing has emerged in recent years as a very lively area in computer science.*" We can see that computer graph has become more and more important in the modern world. Taking a look around, you can find many examples which use this technique to show information. If you have some experience about using the subway, what you firstly do before you get on the train is probably looking at the map and try to find the right path to your destination. These maps often are interpreted as graphs.

There is also a very good example of how graph used in the traffic system



Figure 1.1: Tube Map [3]

which is called "*tube map*" coined in 1931 by Beck [3]. Comparing with the traditional geographic presentation of the tube map, Beck developed a new approach in which all the connections (the tube lines) between each node (station) are straight lines, as you can see in Figure 1.1.

Taking a look at Figure 1.1, this is a tube map of London in 2011. This kind of map does not represent the real geography of the city and traffic, however, as a passenger he or she may not concern about the geography things, what they really concern is to go to the right destination with right path, which in other words the less time consuming. These straight lines can indeed help people to calculate the distance and find their destinations and then save time.

Another typical application of graphs is work-flow. The work-flow could help the managers get the activities and relevant events, eliminate the redundant tasks during the working process and combine the same activities together. Thus, it is useful in improving the efficiency of a job or program. Figure 1.2 shows a simple instance of work-flow graph. We can see that there are several nodes and edges in this graph. It is not hard to relate each event with the activity connected. According to this kind of graph, there is also one kind of graph called "*st-graph*". It contains only one "*source*" node and one "*terminal*" node [2]. When used in business, they separately represent the beginning and the end of one process. In this case, Figure 1.3 will present an example of *st-graph*. We can clearly see the "*S*" and "*T*" node. And the nodes between them are separated into several levels.

## 1.2 Problems

Graphs, especially *st-graph*, has been widely used in many areas. One source one sink model makes it naturally and easily to fulfill the common process in general business. Taking a look at Figure 1.2, it is not hard for us to distinguish between each event and activity as the graph is really clear to see. However, if we got hundreds of events and probably thousands of activities, then using the model as Figure 1.2 shows will become ineffective because of too many edge crossings. Taking a look at Figure 1.3, it is difficult to distinguish from different paths with

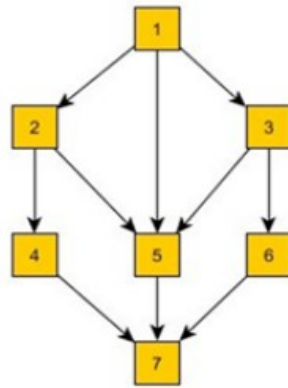


Figure 1.2: Work-flow Graph

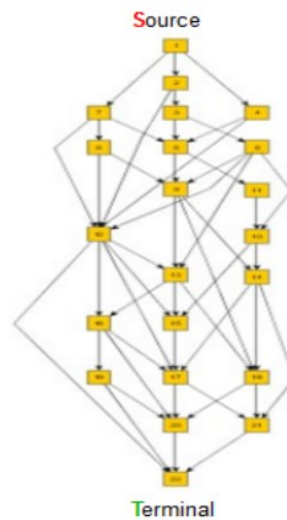


Figure 1.3: St-graph

connection of the nodes in different layers.

According to this issue, there is a good layout of st-graph called planar st-graph [4] shown in Figure 1.4, which has no interleaving between two or more edges. As defined by McKay, B. a "Planar graph" is "a graph that can be drawn in a plane while do not has edge crossing" [5]. Thus if this kind of graph is "st-graph", then we call it "planar st-graph" [4].

Comparing with Figure 1.3 who has lots of overlaps and edge crossings, this kind of graph can give us a better ability to distinguish the nodes and lines. Indeed, we can implement it even with thousands of edges to draw. However, arrow lines always have problem with distance  $C$  its often difficult for people to follow the direction or the path of one line if it is too long or too short, especially when there are many other parallel arrow lines beside it. For example, Figure 1.4 shows an example of a planar st-graph with several long and parallel paths, but actually, we can even draw one specific path cross 100 or more layers. Now the problem is how to find a specific path if we need? For instance how to distinguish

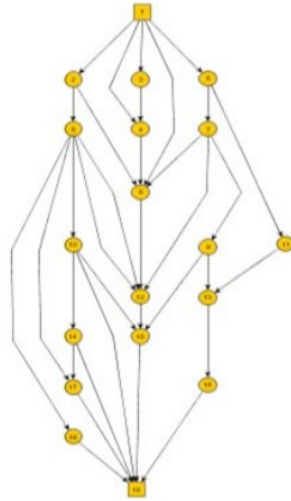


Figure 1.4: Planar st C graph

the different paths from source node to the sink node? Obviously, the result will be even disappointed when the path becomes more and more deep. Therefore, we still need a new layout to make things better. This layout should make a more intuitive way to represent the nodes and edges in order to overcome or mostly reduce the problem of overlap and edge crossing.

### 1.3 Motivations

Taking a look at all the graphs illustrated in Figure 1.2-1.4, it is not hard to see that they both use a separate node to represent an event and arrow lines to represent the activities or relations between each node. Actually, in the field of graph drawing, there are many other ways to draw such kinds of structured graph. One of them called "Treemaps" [6] which uses rectangles to represent the nodes and the relations between nodes will be intuitively recognized by the mappings. Figure 1.5 (b) shows an example of tree maps which represent the relationships between each node in (a). As Mark Bruls et al indicated "the full display space is used to visualize the contents of the tree" [6]. Comparing with (a), the tree map (b) adopts a better topology to represent the relations. As you can see here, the leaf nodes in (a) are presented by the individual rectangles in (b), and the size of non-leaf nodes were the sum of their children nodes. In this case, tree map is much more effective when comes to a large graph.

The features of "Treemap" provided us the motivation that to use shape segments to present the nodes (e.g. rectangle, curves etc) and the relationships between nodes will be represented by the mapping relations. The idea also partially reference I.G. Tollis's idea when he talked about the numbering assignment of "st-graphs" [7] and can be better described in Figure 1.6 from (a) to (b) and (c). It is worth to mention that we are here not using treemap to present the "planar st-graph", instead, we partially adopt the concepts and approaches in treemap to

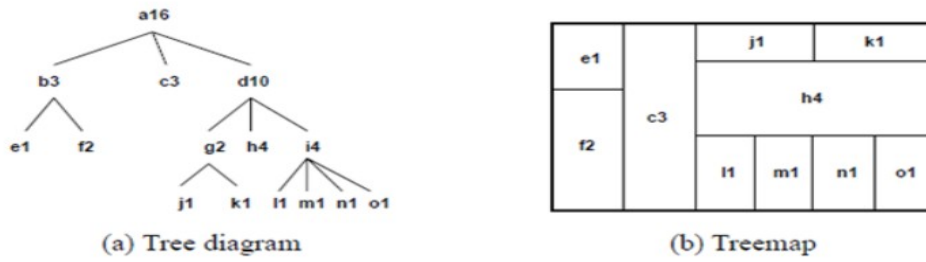


Figure 1.5: Tree map

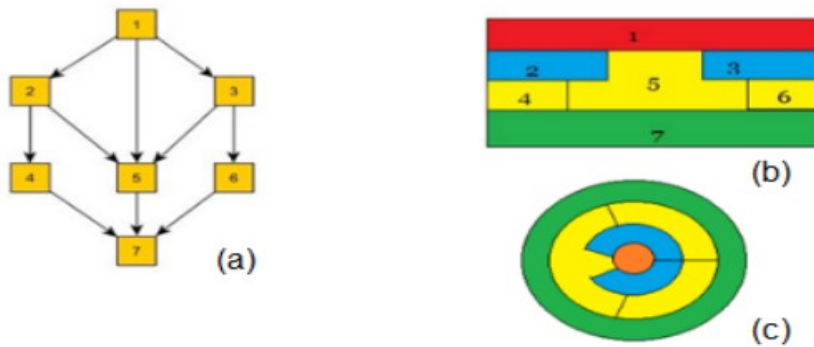


Figure 1.6: Motivations

make the visualization development to generate a new kind of type for presenting "planar st-graph".

### 1.4 Project Goals

Considering the problems of space waste and the difficulties of identifying relations between two nodes in a long-path planar st-graph, in this project, the final goal is to develop a new space-filling technique for the visualization of planar st-graph in order to better fit human intuitions so that the screen space can be maximum utilized and the relations between each nodes can be better represented.

### 1.5 Restrictions

There are several restrictions in this project: a) The research object is very specific graph C "planar st-graph" [4]. In this case, we need the user firstly upload a "planar st-graph" as the input resource, this means the users themselves need to have some knowledge about graph and he or she does know how to make a planar st C graph; b) We use curves to represent nodes, although it can save spaces since there is no edges (links) shown in screen, however, since there is less interactions, when there are too much nodes exist in one layer, it will probably lead to the situation that some of the nodes are too big (means the curve is too long) while others are too small; c) The input is a planar st-graph can not guarantee that



there is no edge crossing when it drawn on screen. In order to do this, we need to do the job of '*Layered drawing of digraphs*' which consists of three main steps: '*Layer Assignment*', '*Crossing reduction*' and '*Horizontal Coordinate Assignment*' [4]. Step 1 and 3 will be done automatically by implementing the relevant algorithms, however, since step 2 is more complicated and it can be seen as an extension part of this project, so we decide to reduce edge crossing manually. However, this can become a restriction since there are always some problems with manual works.

## 1.6 Structure of the report

The whole structure of this thesis will be as follows: The first chapter mainly introduces the background and problems of graphs including: normal work flow graph, st-graph, tree map, space-filling and so on. And the motivation and goal of this thesis study will be formulated in this chapter as well. After that, in the "*Related Works*" chapter, we will try to cover all the related concepts and ideas and techniques in this project. E.g. Info-Vis, planar st-graph and graph drawing approaches and so on. This chapter can give readers a comprehensive view of what elements are contained in this thesis; then come to the main part of the project: visualization and implementation. These two parts will mainly talk about how this project running from an initial text-based file to the final circle-based layout of new space-filling techniques to show the planar st-graph. At the end, discussion and future work will be conducted to summary the whole working process and learning outcomes as well as the improvements of this project.

## 2 Related Works

This chapter is about the introductions and explanations of concepts and techniques that will be used in this project. For example, what is the subject area of information visualization, or what are the concepts and characters of planar st-graph and so on.

### 2.1 Information Visualization

Information visualization is an area in computer science which is used for transferring data and information into visual forms [8]. In information visualization, we can solve problems in a much intuitive way based on humans' natural perception. Visualization combines computers and our brains together to help people to analysis and understand data with different methods and techniques in computer graphic.

Information visualization works in many areas such as biology, medicine, geography, business and so on. Comparing with original computer graphic, Info-Vis has many advantages. Static and dynamic data can be available for many people and groups. Patterns and correlations can be discovered and using Interactions to let user to manage or operate different views on the screen to get different results. The typical example of that is the Info Vis Reference Model as Figure 2.1 will show the model. In this figure, we can see that there are two basic part "DATA" contains "raw data" and "data table" and "visual form" contains "visual structures" and "views". It is very clear to see how raw data is transferred into views with "data transformations", "visual mapping" and "view transformations" and human interactions between each part.

According to this model, in this thesis, the raw data will represent the nodes and edges which present relations and directions between nodes. And we select the graphml format as data table to format the raw data. As introduced in Section 3.1, the graphml file contains the information of nodes ID and edges which contains the "source" node ID and "target" node ID, therefore a graph with "graphml" format has already been well structured. Then we store the information of nodes and edge (the relations between nodes) in different Java vectors. When all these information are collected, we can start by assigning the layers and x coordinate positions for each node and drawing links according to the "edge" information in graphml file. So far, we already make the graph in a structured way. After that, we choose a space-filling technique [8] as the visual structure, and the final representation only consist of circles and curves.

### 2.2 Planar st-graph

In the Introduction chapter, we have already mentioned the concept of planar st-graph briefly. In this section, we will bring more and detail explanations about the relations between "st-graph", "planar graph" and "planar st-graph". What is

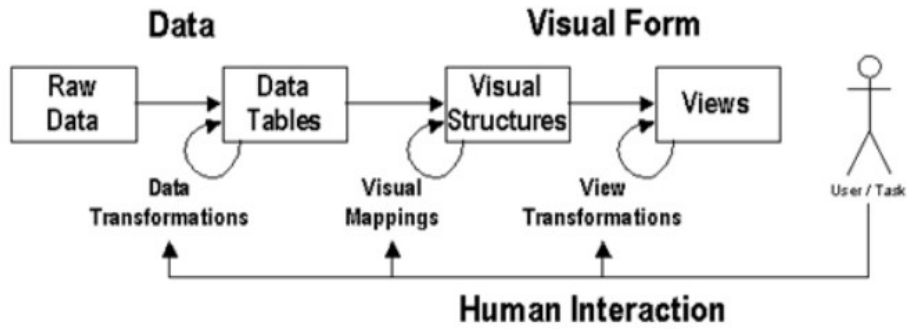


Figure 2.1: Info Vis Reference Model [8]

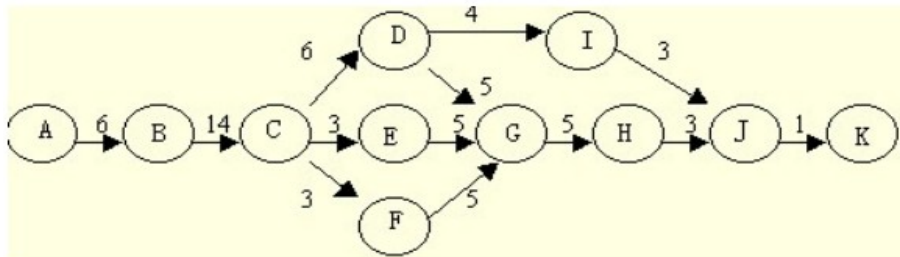


Figure 2.2: PERT example

more, we will also bring some examples of how planar st-graph are adopted in the real world to better explain its features.

### 2.2.1 St-graph

We category graphs in many ways, for example: "planar" or "non-planar"; "directed" or "undirected". Here we will introduce one kind of graph called "St-graph" [9]. The st-graph also has been widely used in our lives. Some typical instances like: PERT graph in network work flow model. PERT is short for 'Program/Project Evaluation and Review Technique' [10]. It is a technique that makes and evaluates plans by using network analysis technique. It can coordinate each event in a whole work plan, arrange manpower, material resources, capital and time in a proper way so that speed up the completion of one project. It is an important method and means in "modern project management"

Taking a look at Figure 2.2, this graph consist of many nodes represent events and arrow lines connect each nodes represent activities between two events. Mapping to the st-graph, A is the source node means beginning of one program and K is the sink node represent the end of program. All of the arrows must point at the same direction. And it is also a planar st-graph since it has no edge crossings.

### 2.2.2 Planar Graph

As we know, a graph consists of a set of vertices, and a set of edges each joining two vertices. In this case, a "planar graph" is a graph which can be embedded in

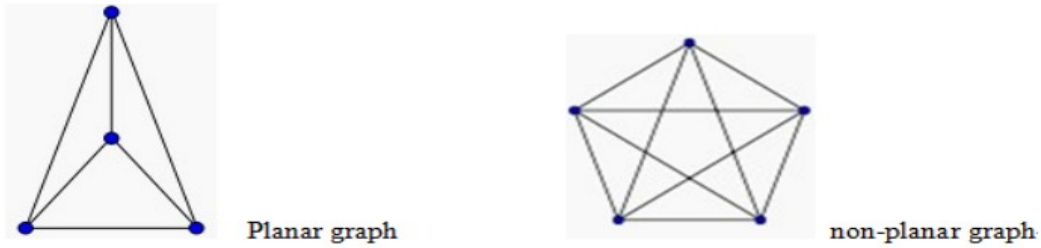


Figure 2.3: Planar and non-planar graph

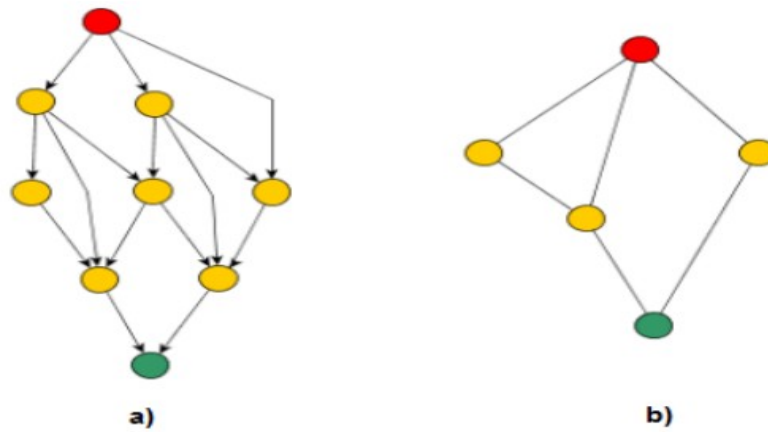


Figure 2.4: Straight line drawing

the plane so that no two edges intersect geometrically except at a vertex to which they are both incident [11]. And increasingly, the concept of "*planar graph*" has attracted many computer scientists due to its plenty applications [11]. You can see the difference of planar and non-planar graph from figure 2.3.

Nowadays, the automatic drawing of planar graph has bring intense interest in related areas like "*computer network*", "*VLSI layout*", "*information visualization*" etc., and a number of drawing styles are coming out consequently like "*straight-line drawing*", "*straight-line grid drawing*". [12] The idea of "*straight-line drawing*" approach is the drawing way that draw each node as a point and edge as straight - line without edge crossing, and in this case we also need to distinguish whether the graph is a "*planar st-graph*" which has direction as shown in Figure 2.4 a) or undirected planar graph as it is in Figure 2.4 b) since the method for drawing these two graph is different; the "*straight-line grid drawing*" method as shown in Figure 2.5 is based on an integer grid so that each node will be drawn on the specific grid point, so that it can eliminate edge crossing by putting nodes to the appropriate positions. [12]

Here in this project, we will adopt the approach of "*straight-line drawing*" in Figure 2.4 a). The detail information concluding drawing algorithms and ideas will show in Chapter 3.

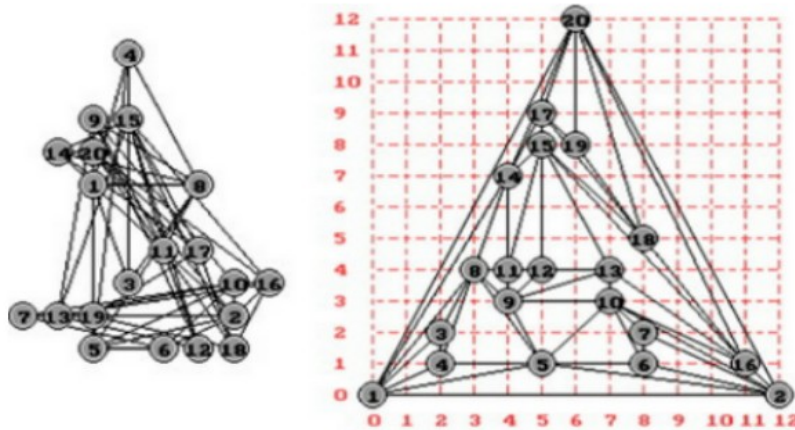


Figure 2.5: Straight line grid drawing

## 2.3 Space-Filling

In Chapter 2.1, we have introduced that in information visualization, there are many techniques solving different problems. Here, in this small section, we will introduce one of the most popular space-filling techniques: tree maps as well as radical space-filling in order to motivate ideas.

### 2.3.1 Treemap

Before talking about Treemaps, we need firstly introduce the concept of hierarchical structure. As T. Nishizeki et al described, "A large quantity of the world's information is hierarchically structured: manuals, family trees, computer programs ..." [13]. Indeed, with the ever increasing data and information storing in our PCs, mobiles and other storage, a kind of hierarchical structure like the file system in Windows can greatly increase our working efficiency. However, T. Nishizeki also pointed out, "Most people come to understand the content and organization of these structures easily if they are small, but have great difficulty if the structures are large" [13].

There are three traditional methods for presentation of hierarchical structured information: *listing*, *outline*, and *tree diagram*. *Listing* is a very common approach of showing hierarchy, and it does can provide structured information when it listing the entire structure with explicit path [14]. However, it is inefficient since users need to parse the path manually; *Outline* approach can provide both structure and content at the same time, but the limitation exists that users can only see few lines of information at one time [14]. Actually, both of these two methods are "out of fashion". Nowadays, when talking about hierarchical structure, most people will think about using tree diagram. Henry, T.R. And Hudson, S.E. [15] introduced this approach as "... it has traditionally sought efficient and esthetically pleasing method for the layout of node and link diagrams... it is excellent visualization tool for small trees". The comparison between outline and tree diagram approaches

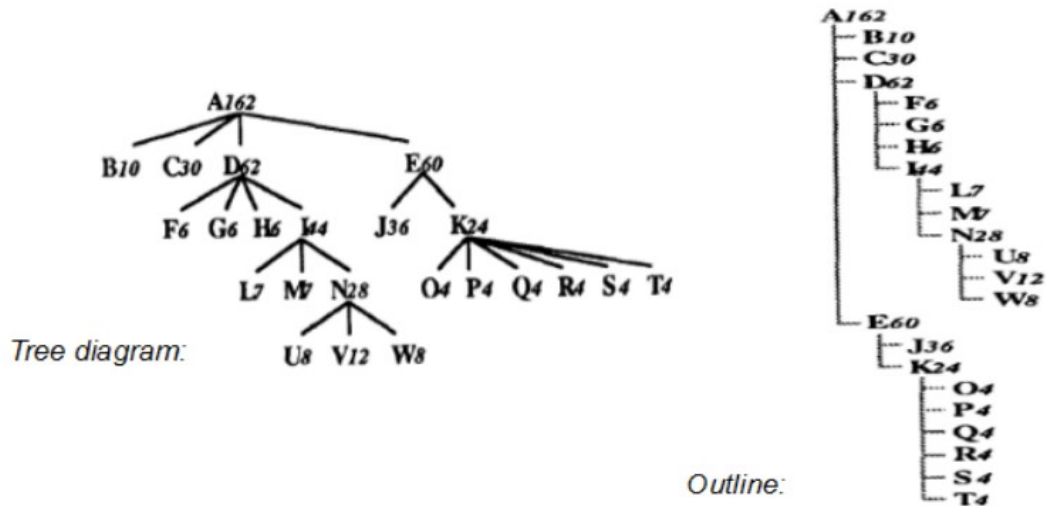


Figure 2.6: outline and tree diagram

is shown in Figure 2.6.

However, even a tree diagram has limitations: a) poor use of available spaces. For example, in the tree diagram of Figure 2.6, almost half space (pixels) are used as background, in this case, when graph become really large, the adequacy of information shown in this tree will be low due to overlaps of nodes and edge crossings; b) poor quality of content since we need to put each node's content as a text standing in the position as the node should be. Therefore, we introduce the idea of *Treemap* here to see what are the difference and advantages of Treemap comparing with traditional approaches.

We can define the main limitations of traditional hierarchical structured approaches are space and content. While in Treemap, all these two problems can be solved. Firstly, Treemap use 100% of the designated display space, secondly, it allows user to see both the structure (e.g. Depth) and content (display properties such as color mapping) [14]. Taking a look at Figure 2.7 which shows the corresponding Treemap of the graph in Figure 2.6, we can find there is no gaps between two rectangles, i.e., there is no space waste. And the content can also be better represented in nested shapes with color mapping.

### 2.3.2 Radial Space-Filling

When the Treemap was coined, there appeared many Treemap layout algorithms for that, such as: slice-and-dice, Cluster, strip, etc [15]. All of these algorithms using rectangles to represent nodes in the original node-link graph. However, we can also use radial segments instead the rectangles. There also several examples of this in information visualization, such as: sunburst, angular details, interring and so on [16], as shown in Figure 2.8.

This example is also the inspiration of my project. Using radial segments to present nodes, by the special layout of radial circles, it is easier to arrange the

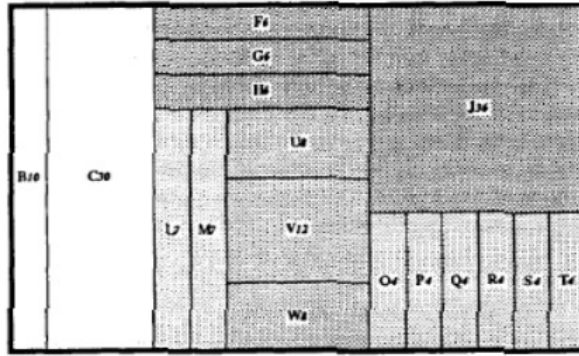


Figure 2.7: Treemap



Figure 2.8: Radial space-filling [16]

proper layer for each node and also easier for users to distinguish different layers comparing with the rectangle layout.

## 2.4 Parameterized st-Orientations

I.G.Tollis and C.Papamantou in their article called '*Applications of Parameterized st Orientations in Graph Drawing Algorithms*' [7] described a new way to represent planar st-graph as shown in Figure 2.9. They use red rectangles to represent the '*source*' node and green rectangles to represent '*sink*' node. The yellow rectangles between them are '*middle*' nodes and we can see they are assigned into different layers and these are typical structured graphs.

## 2.5 Processing

To do the project, we need to use one of the most popular languages JAVA to implement algorithms. Considering the drawing of nodes, lines, and radial segments, we choose the Processing API to implement this part. "Processing is an open source programming language and environment for people who want to

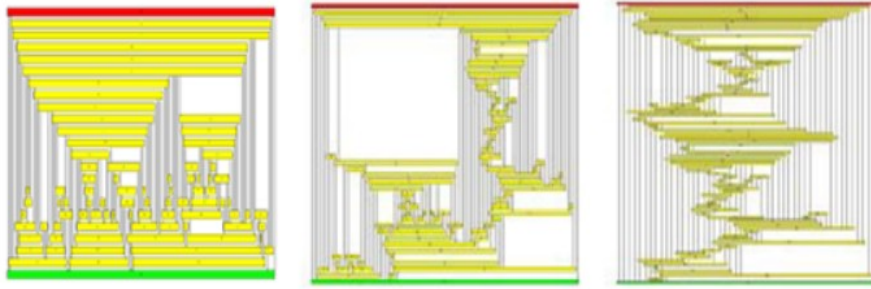


Figure 2.9: st-orientation in graph drawing algorithm [7]

create images, animations, and interactions.” [17] Since this project do not need to use too much visual patterns, using processing is much efficient because it is easy to learn and use.



### 3 Visualization

The main ideas of visualization for this project will be introduced in this chapter. It includes the conceptual design of user interface as well as the project running flows. Moreover, visualization patterns from the original planar st-graph data source to the final representation of space-filling layout will be described in detail.

#### 3.1 Data Source

As in Chapter 2 referenced, the main process of this project is to firstly input a planar st-graph, in this case, a *graphml* file [18]. After eliminating the edge crossings, then convert it into a new diagram using the spacing-filling technique. Here, we choose *graphml* file as the input source. The reason why we choose “*graphml*” is because it is a comprehensive and easy-to-use file format for graphs. It consists of a language core to describe the structural properties of a graph and a flexible extension mechanism to add application-specific data. Its main features include support of almost all kinds of graphs include directed, undirected, hierarchical graphs and so on [18]. Figure 3.1 a) shows us a simple example of *graphml* file with basic elements.

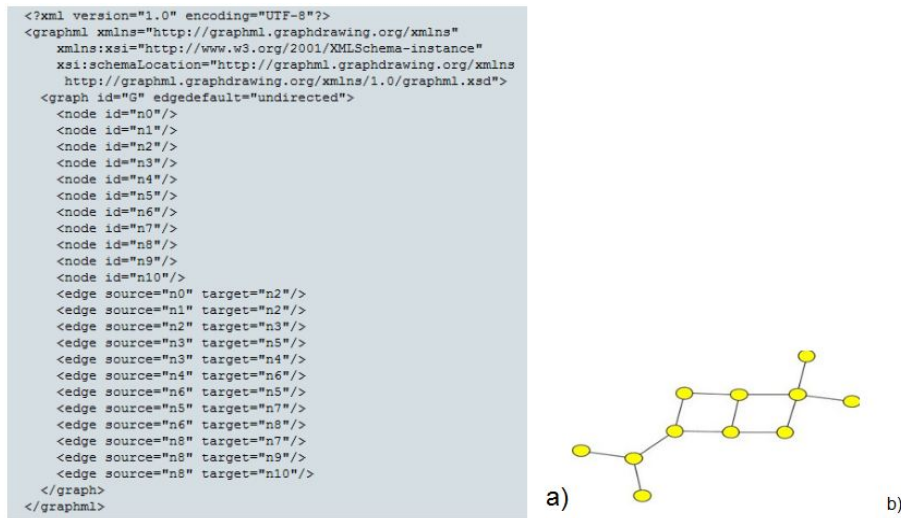


Figure 3.1: graphml file

Graphml is based on XML; taking a look at Figure 3.1 you will find it is similar with a common XML file if you were familiar with. The difference is the declaration part in order to tell that this is *graphml* format. The father node ‘*graph*’ has two children ‘*node*’ and ‘*edge*’, attribute of ‘*node*’ is to identify each single node in the graph, while ‘*edge*’ element has two attribute ‘*source*’ and ‘*target*’ to identify the beginning node and the ending node of this edge respectively. The result of this graphml will be shown as Figure 3.1 b).

The advantage of using *graphml* file as input is: It is based on XML, so, it is easy to be parsed. Comparing with other file format for graphs, *graphml* has

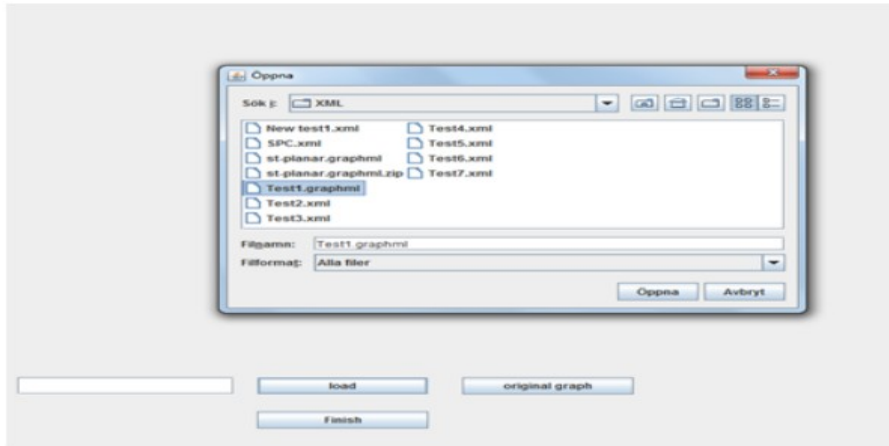


Figure 3.2: Initial View (Input)

a better structure which made it easy to be viewed and modified. Example of Figure 3.1 only shows a common undirected graph. Actually, in this study, the initial graph we need is so called st graph which means this graph only has one source node and one sink node, and this graph also need to be conducted in a directed structure from father node to children nodes, as shown in Figure 2.2. So, in this case, in the initial *graphml* file, we need to define a root node "n0" to be the source node, as well as the last node to be the sink one. What is more, there is no leaf node here or in other word all nodes need to have their own father node(s) and children node(s). So all the content in Figure 3.1 a) can be seen as the requirement for the content of input graphml file.

### 3.2 Initial View

The initial view of this program is a Java applet which contains several command buttons. Users need to first upload the graphml file as the input data by clicking 'Upload', when they finish uploading, the path of this file will be shown in the left text-box and then click button 'original graph' the corresponding original planar st-graph will be shown in the left-top screen.

In order to make it possible to transfer the input graph into space-filling layout, users need to change some position of nodes in some specific layers to eliminate the edge crossing. This can be down by just click on one node and drag it to another node in the same layer. After that, click 'Finish' button and then the new layout will be shown in the right part. This process will be shown in Figure 3.2 and Figure 3.3.

### 3.3 Layer Assignment and X Coordinate Assignment

Given by the initial graphml file as described before, how can we make it into a graphic-based structure and show it out? This is one of the most important steps in this project-layer assignment. In this section we will cover the main idea of the



Figure 3.3: Initial View (Original Graph)

corresponding algorithm as well as when we finish putting all the nodes in the right layer, how can we assign the X axes for each node in the same layer.

In Section 1.1, we have introduced the concept of graph and its categories. In this project, the graph is not only a directed but also an oriented graph. Taking a look at the three figures in Figure 3.4. If we count from left to right. We will find that the first figure is exactly what we call it "*undirected graph*", here we can say the '*blue*' and '*yellow*' nodes are the children nodes of the '*red*' node, but we can also see '*yellow*' node as the father node of both '*red*' and '*blue*'; when comes to the second one, we can already see the difference with the former one that it has directions between each node, which means there are relations exist among these nodes. However, given the nodes and edges as the second figure shows, the output diagram of this graph should be different in this project. The reason is exactly due to the concept of "*planar st-graph*", in st graph, the depth of each node is also important since the graph itself should have a direction as you can see in both Figure 1.3 and Figure 2.4. Therefore, in Figure 3.4 b), if we only abstract the area in rectangle, i.e., only focus on Yellow node, we can say the the orientation is from left to right, however, when it comes to the entire graph, the direction is not unique. In this case, we need to reconstruct the structure of this graph in order to identify the direction. Figure 3.4 c) shows the idea, here we use an algorithm to make sure that all the "*target*" nodes' depths should deeper than the "*source*" nodes' depths. The yellow node is both the "*target*" node for the red and blue node, while blue node is the "*target*" node of red one, therefore, we need to put yellow node in layer 3, as you can see in c), now the edges in both two layers (layer 2 and 3) are top to bottom.

### 3.3.1 Layer Assignment

As long as we defined the format of the input graph, we can start to do the algorithm to transfer it into a 'real' graph. For a planar oriented graph, all of the directed edges should point to the same direction, such as 'top to bottom' or 'left to

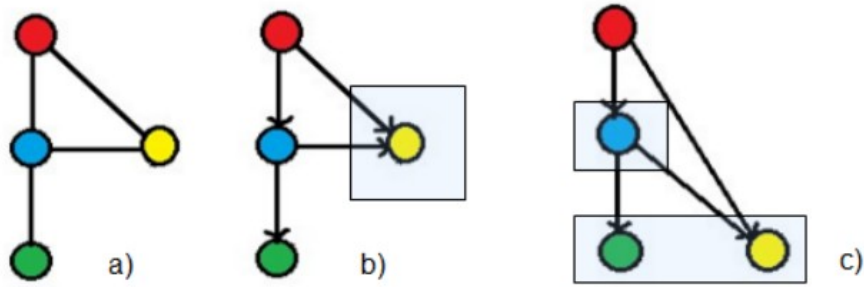


Figure 3.4: The direction of graph

right'. We can not mix the directions in the graph. For example, the second graph in Figure 3.4, there are two directions: top to bottom and left to right, while for the last one, the relations between each node is the same as those in the second graph, but all of the edges' directions are top to bottom, and thus we can say this graph's direction is "top to bottom", so this is an oriented graph.

Now the problem is: given by an oriented graph, with the nodes and relations between them, how can we know the proper layer for each node? To make a specific analysis, we take the oriented graph (OG) in figure 3.4 as example. This graph is a simple oriented graph. And it is obvious to see that 'red' node has two children 'blue' and 'yellow' while 'blue' node also has two children node 'green' and 'yellow'. If we give each node a set to store its children nodes, the result of OG is:

*Red: Blue, Yellow*  
*Blue: Green, Yellow*

And this is also the first step in this algorithm - putting all the children nodes into the father node's set. This procedure can be easily achieved by parsing the graphml file since the 'edge' attribute can help us find out the relations between 'father' and 'children'.

After finishing the first procedure, the layer assignment for each node can be formulated based on the nodes and the nodes in their sets. Firstly, we set the depth of each node in this graph to 0 (since we define the number of first layer as layer 0), then we analysis from the beginning node in this case - red, and we find that this node has two children nodes: blue and yellow, so we know that these two nodes' depth at least should add 1 depth, thus 'depth ++' for blue and yellow. Again we find node blue also has two children: yellow and green. Here comes out a trick situation, we can again let node yellow add 1 depth and therefore its depth will become 2 which mean it will be assigned into layer 2; however when comes to the node 'green', we can not simply use 'depth ++' since the outcome will become that the depth of green node is 1, but we all know it should be 2. In this case, we decide to use 'replace' approach instead of 'add' approach. The

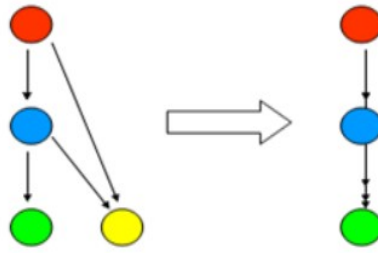


Figure 3.5: X-coordinate motivation

main idea is: we change the 'green' and 'yellow' nodes' depth into "blue.depth +1". And this process will continue until we reach to the last node.

In this case, we defined an algorithm corresponding to the analysis in this section.

- 1: We have two empty sets *set1* and *set2*.
- 2: *Initial set1, set2;*
- 3: *set1*  $\leftarrow$  all nodes; empty *set2*;
- 4: **for** (node *x* : *set1*): *Initial set A*;  $A \leftarrow (x.getSourceNode \cap \textit{set1})$ ;  
*set2*  $\leftarrow$  *set2*  $\cup$  *A* **end for**;
- 5: *set1*  $\leftarrow$  (*set1* - *set2*);
- 6: **for** (node *x*: *set2*): *x.depth* ++ **end for**;
- 7: *set1*  $\leftarrow$  *set2*;
- 8: **while** *set1.IsEnd* == *false*; loop from 4 to 6.

The implementing part of this algorithm will be shown in the Section 4.1.1. So far we have covered the process of layer assignment, but this is not sufficient for drawing the graph into the screen. Because 'green' and 'yellow' are both in layer 2, but this can only mean that their Y axes coordinates (2 depth) are different with 'red (0 depth)' and 'blue (1 depth)'; however, as shown in Figure 3.5, without additional steps of x-axes coordinate assignment for these two nodes, they will be drawn at the same place. Therefore, here we need to do the x-axes coordinate assignment in order to distinguish the nodes' positions in the same layer.

### 3.3.2 X-coordinate Assignment

When we finish all the tasks of layer assignment, we can already get the information of what are placed in which layer. So here we define a 'Layer' class and give it a Vector to store every node in this layer. Hence, despite the relations of nodes in different layers, here we can only focus on each specific layer and specific nodes in this layer. In order to doing this, we initialize every node's *x* coordinate to 0. After all of the nodes been assigned to every specific layer, using a simple "for" loop for each layer can easily assign *x* coordinate to every node in this layer.

It is worth to mention that: what we finally gain here is only the node's depth (layer) and its position in this layer (e.g. layer (2). Position (1)). When we want to show the graph on screen, we need to calculate the X coordinates as well as Y coordinates for each node. And here the key factor that needs to be taken into concern is the 'scale'. I.e. with the increment of the number of layers in one graph, the increment of Y coordinate from root to sink should be small, and the same as the increment of X coordinate for nodes in each layer from left to right. And what's more, in order to make it possible for people to recognize the nodes and edges, we also need to define a threshold to limit the minimum scale.

So far we can say that a graph based on the input *graphml* file can be shown out. Still we can not guarantee that there is no overlap or edge crossing inside this graph although we know it should be. The reason is because the X coordinate, or to be simple, the position of each node in its layer. We can just take figure X as example, although we finish the X coordinate assignment, for 'green' and 'yellow', there is still one possible situation that shown in Figure 3.6.

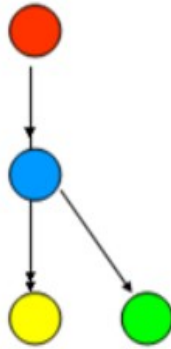


Figure 3.6: X-coordinate assignment motivation

In this case, it is difficult for people to recognize that 'red' is the father node of 'yellow'. Therefore, we need to eliminate the edge crossing and overlap by firstly adding dummy node and then change position of nodes in the same layer.

### 3.3.3 Creation of Dummy Nodes

Dummy node is an empty template node that you can use to build new nodes [4]. In order to eliminate edge crossing, adding dummy nodes is important. There are several algorithms for reduce edge crossing, most of them are based on dummy nodes. In this project, dummy node is also important. We can just take Figure 3.7 as example.

Actually, this figure was taken from one instance of this project when it starts running and input a *.graphml* file. The relationship of this graph should be show as below:

**N0:** N1, N2, N3

- N1: N3
- N2: N3, N5
- N3: N4, N5, N6
- N4: N6
- N5: N6
- N6: N7

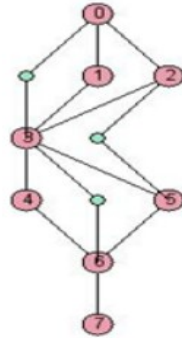


Figure 3.7: Dummy Node

Therefore,  $N0$  should connect directly  $N3$ ,  $N2$  should connect directly to  $N5$ , however, if we do that, then it will become very difficult when we want to reduce edge crossing: change the position of edges is difficult than change nodes. Thus we put a small node between  $N0$  and  $N3$ . By adding dummy nodes, the length of each path between two nodes will become the same: one layer distance. According to the algorithm of edge crossing reduction, [4] this is the key step since the algorithm its method is to compare the nodes' and edges' situation between two layers. The dummy node will be generated automatically according to the algorithm shown in next section, but the edge crossing eliminating process will be conducted manually as you will see later.

### 3.4 Eliminate Edge Crossings Manually

In this thesis, we only discuss planar st-graph. A planar graph should have no edge crossing, but after we input a file, this does not mean that the output is a standard planar graph without any edge crossing. And when doing the space-filling part of this project, the requirement is also eliminate the edge crossing as explained in Section 2.2.

We accomplish this function by adopting *Processing API* and the ideas is as follows. As you can see in Figure 3.8, the users need to click on the specific node area (the area represented in rectangle) and drag it to another node's area from which they want to change the positions. By doing this, two properties are changing: If these two nodes have a common father node, then their index in the father

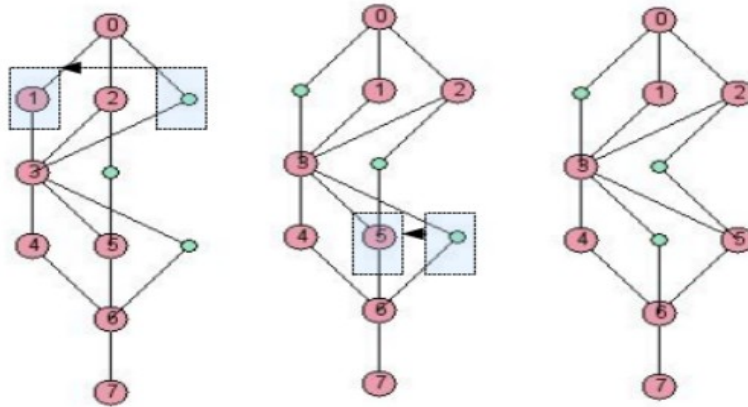


Figure 3.8: Edge Crossing Eliminating

node's "children node set" will change; and their index in the specific layer will change as well.

In this example, we start by firstly 'drag' *Node 1* to the dummy node in its layer which is a small green node, and 'release' the mouse, then we can see that the position of these two nodes had changed, and then we drag *Node 5* to the dummy node in its layer in the same way, and at the end the result will have no edge crossing.

### 3.5 Drawing Arcs

What we have down so far is the basic and also one of the core processes of this project – basic graph layout. And the rest part will be another core part: space filling visualization. As you already see in the introduction part, what we are going to do is to use arcs to describe the nodes and the relations in each node will be presented by the projections or connections of each arc.

The general idea is to put a circle in the middle to present the source node in *layer 0*, and then we will assign several curves surround this circle to present its children nodes. What's more, the small gap will be generated if there is any dummy nodes exist in this layer. And continue this idea layer by layer. We can take the example of Figure 3.9 and illustrate the evolution from the source circle to its connected nodes (arcs) in order to get the whole idea.

We can clearly see the gap here representing the dummy node. Actually, we do not really 'draw' this node out, and instead we assign the start and stop parameters of the curve for this node in order to 'leave' this space out. i.e. for the *layer 1*, we will draw from *node 1* instead of the dummy node, and since we had already assigned the stop parameter of this dummy node, therefore, we can take use of it to define the start point of *node 1*. the process of how to implement this idea including how to assign the parameters and which kind of situations must we take into consideration will be explained in detail in Chapter 4.



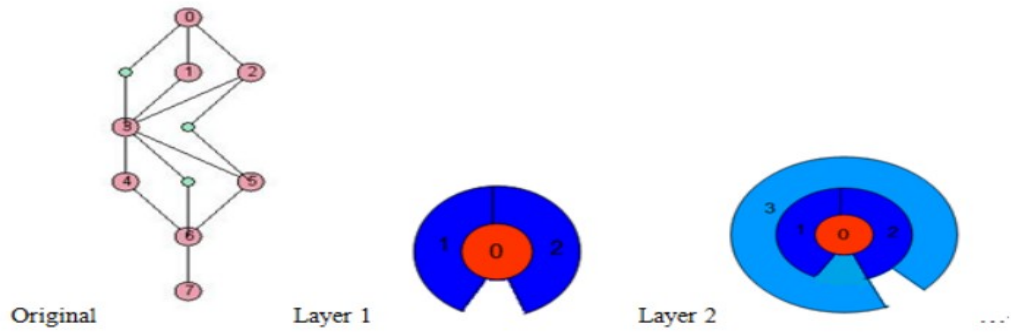


Figure 3.9: Motivation of Drawing Arcs



Figure 3.10: Project result

### 3.6 Interactions

Since this project mainly focuses on the new idea of space-filling techniques for graph, therefore, what we mostly focus on is the algorithm for new layout, which means we can not add too much user interactions on it. There are two features can be seen as interactions on new space-filling graph: labeling and brushing. As you can see in Figure 3.10, when we put the mouse on one specific curve, the corresponding node ID will shown on the white tooltip, at the same time, on the left side the original graph, the homologous node will be highlighted with other color.

## 4 Implementation

This section will mainly talk about how the ideas or algorithms in Chapter 3 were implemented. This will cover two main parts:

Part 1: "Original st-graph visualization" which also concludes four small parts: 1) *layer and x-coordinate assignment*, 2) *generating dummy node*, 3) *edge crossing reduction (manually)* and 4) *graph presentation*.

Part 2: "Space-filling visualization" which conclude mainly three parts: 1) *calculating and assigning parameters to each node as start position and end position of arcs as well as for the dummy nodes*, 2) *calculating the number of layers to determine the length of the gap between two layers* and 3) *drawing the arc and start and end lines of each arc*.

### 4.1 Original Planar St-Graph Visualization

In order to let users better distinguish the differences between traditional approach of drawing planar st-graph and the new space-filling technique, we will firstly represent the original planar st-graph visualization based on the input data source from the user side. This section therefore is to describe the implementations of how to transfer the input data source into graph representation.

#### 4.1.1 Layer and X-coordinate Assignment

This section will focus on how to assign nodes to the right layer and the right place in this layer based on the ideas talked in Chapter 3. In this project, we have a class called 'LayerAnalyzer' responsible to do the job of assigning and changing nodes' positions, see Figure 4.1.

Here the function named 'DoLayer()' is used to assign nodes to each layer. And there is another class with data structured JAVA Vector to store information of nodes and their relations which called 'LayerSet' shown in Figure 4.2. The Boolean function called 'Add()' shown in Figure 4.3 together with 'Find()' function here in this class is used to add new nodes into a set of one specific node. E.g. in Figure 4.9, *Node1*, *Node2* and *Node3* will be added into '*Node0*'s *LayerSet*, this means they are the children nodes of *Node0*. In order to implement the algorithm, the class of 'LayerNode' is also necessary to store and manage the information in each node which consists of the information of *depth*, position and the function of increase depth, return ID and so on. And for each *LayerNode X*, we assign it a *LayerSet - 'Set'* to store its children nodes as well as a *LayerSet - 'Fathernode'* to store its father nodes.

Now we come to the 'DoLayer()' function (shown in Figure 4.5) in class *LayerAnalyzer*. This is the implementation of the algorithm mentioned in Section 3.4.1. We start by firstly putting all nodes in *LayerSet .d1*, go through this set and



```

public int Find(LayerNode _node) {
    if(_node==null) return -1;
    int i = 0;
    for(LayerNode x : vec) {
        if(_node==x) {
            return i;
        }
        i++;
    }
    return -1;
}

public boolean Add(LayerNode _node) {
    if(Find(_node)>=0) return false;
    vec.add(_node);
    return true;
}

```

Figure 4.3: Add (LayerNode \_node) function

```

public LayerSet UniteSet(LayerSet _set) {
    for(LayerNode x : _set.GetSet()) {
        Add(x);
    }
    return this;
}

```

Figure 4.4: UniteSet function

find each node's own set which contains its children node and UniteSet (shown in Figure 4.4) with LayerSet **d2** and give them *depth* ++, keep doing this process until **d1** is end and now each node has its own *depth*. We will then get the *depths* and use them to assign nodes to right layers.

When the 'Layer Assignment' was done, all of the *LayerNode* X in the 'list' (which is a *LayerSet* to store all the *LayerNodes*) will have their own *depths* which means we had already assign them to the proper layers. And then the next step is to assign X- Coordinate to each layer. This step is simple: we had conducted a class called 'Layer' which used to store information of layers as shown in Figure 4.6. We use the user function 'assignxcoord()' to implement this step by using two 'for' loops functions to go through all the layers and the nodes inside each layer in order to finish the X - Coordinate assignment. And changing the coordinate value was achieved by the function in class 'LayerNode' called 'IncreaseXcoord (int num)'.


So far, we can already draw an original '*planar st-graph*' on screen since we have all the nodes' axis positions. the next step is to eliminate the edge crossing and this will be divided into two main parts: *Generating dummy node* and *Change nodes' positions*.

```

public void DoLayer() {
    LayerSet _d1, _d2;
    _d1 = new LayerSet();
    _d2 = new LayerSet();
    for(LayerNode x : list.GetSet()) {
        _d1.Add(x);
    }
    while(!_d1.isAllEnd()) {
        for(LayerNode x : _d1.GetSet()) {
            _d2.UniteSet(x.GetSet());
        }
        for(LayerNode x : _d2.GetSet()) {
            x.IncreaseDeep();
        }
        _d1 = _d2;
        _d2 = new LayerSet();
    }
}

```

Figure 4.5: DoLayer () function

 Layer
 













-  vec1 : Vector<LayerNode>
-  id : int
-  ycoor : float
-  Layer(int, float)
-  GetLay() : Vector<LayerNode>
-  Add(LayerNode) : void
-  SetYcoor(float) : void
-  GetID() : int
-  GetYcoor() : float
-  test(LayerNode, LayerNode) : int
-  exchangex(LayerNode, LayerNode) : void
-  AssignYcoor() : void

Figure 4.6: Layer.Java

### 4.1.2 Generating Dummy Node

In the class of 'LayerAnalyzer', the function 'DoAssign ()' is used for generating dummy nodes. Dummy nodes' information also stored in *LayerNode*, the difference is its type is 'FLAG.DUMMY'. Here the general idea is: we firstly define an *Enumeration* set 'e', and we value the elements *x* (*LayerNodes*) in the list to e, and then we will check the children nodes in each element *x* to see if their *depths* are explicit one deeper more than *x*'s *depth*. If not, we will 'cut' the edge and insert the dummy node between them. the detail code of this procedure can be see in Figure 4.7.

```
public void DoAssign(){
    e = list.GetSet().elements();
    for(int c=0;e.hasMoreElements();c++){
        LayerNode x = e.nextElement();
        int temp =20;
        for(int i=0;i<x.GetSet().GetSet().size();i++){
            int num = (x.GetSet().GetSet().get(i).GetDeep()-x.GetDeep());
            if(num>1){
                temp++;
                LayerNode node1 = x.GetSet().GetSet().get(i);
                x.GetSet().GetSet().remove(i);

                LayerNode _node = new LayerNode(temp,LayerNode.TYPE_MIDDLE,LayerNode.FLAG_DUMMY);
                i--;
                _node.setdeep(x.GetDeep()+1);
                _node.GetSet().Add(node1);
                x.GetSet().Add(_node);
                _node.getfathernode().Add(x);
                list.GetSet().add(_node);
            }
        }
    }
    //end Dummy
}
```

Figure 4.7: DoAssign () function (For dummy node)

### 4.1.3 Edge Crossing Reduction

Since we already have the dummy nodes, we can start the edge crossing reduction processing. Now what is needed to do is to find two nodes in one layer that their X C coordinate positions need to be changed. This is done with user's own perspective and experience, which means they should know which kind of changing can finally eliminate the edge crossing. There is a function in 'LayerAnalyzer' class called 'changeposition (int layer, int from, int to)' of which there are three parameters need to be initialized: the layer, and the two nodes' index in this layer which there positions need to be changed. The idea is also simple: Form a new temp LayerNode C to store one of the two nodes, e.g. A, and delete A, put B into A's position and then put C to B's original position. And what is more, in order to make the changing process more fluently, we will embed this function into the 'mouseReleased ()' function in

```

public void mouseReleased(){

    analyzer.GetLayer()[layer].GetLay().get(to).setxcoor(changexcoor);
    from = (mouseX-20)/40;
    int size = analyzer.GetLayer()[layer].GetLay().size()-1;
    float maxxcoor = analyzer.GetLayer()[layer].GetLay().get(size).getxcoor();
    if(mouseX > maxxcoor)
    {
        from = size;
    }
    float temp = analyzer.GetLayer()[layer].GetLay().get(from).getxcoor();

    if(temp < tempxcoor){
    dochange(layer, from, to);
    if(_test != null)
    {
        _test.dochange(layer, from, to);
    }
    }
}
}

```

Figure 4.8: mouseReleased () function

the 'DrawArc' class as shown in Figure 4.8, this Processing function will be called every time when mouse is released.

#### 4.1.4 Original Graph Layout

This project adopts *Processing* API as the main tool to draw graphs. After finishing the processes of layer and X-coordinate assignment, each node should have its own axis position which contains both X and Y Coordinates, and together with the function in *Processing* called 'ellipse (x-coordinate, y-coordinate, width, height)', we can now easily draw the nodes on screen with circles. And the edge between each node will be drawn with the function of *line* (X, Y) simply. We distinguish Nodes (Red) and Dummy Nodes (Green) by the different color and size of the circles.

## 4.2 Space-filling Visualization

When the edge crossing reduction process has been done, the next step will be the presentation of Space-filling layout. And this is one of the key and most difficult parts in this project.

The class called 'Test1' is used to present the Space-filling graph layout, and the function named 'assignarc ()' is the main function to assign arc parameters to each node. Again take Figure 4.9 as example, and we will explain the whole idea of the visualization process. To draw the arcs we use the function 'Arc (x, y, width, height, start, stop)' which taken from the *Processing* API as well. Here

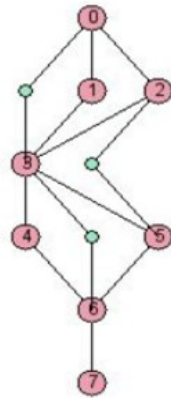


Figure 4.9: An Original Graph

the  $X$  and  $Y$  will both be 0 since all the arcs' center axis is 0; the width and height will depend on the amount of layers of the input graph; and the key parameters here is 'start' and 'stop'. Dummy node is exact not a node, which means we do not need to draw it on the screen; however, it can be one factor to calculate the parameters for Nodes.

Figure 4.9 is a planar st-graph without edge crossing. Now we need to transfer it in to Space-filling layout. Based on the idea in Section 3.5, the **Node 0** will be the central circle surrounding by **Node 1** and **Node 2** and **Node 3**. Since in this case, we also need to concern about the dummy node. We assign a position or a range for it, but we do not draw it.

In order to interpret the idea, we need to reuse Figure 3.10 to illustrate the ideas and issues that need to be concerned during the implementation. As you can see in Figure 4.10, here we use a 'double' variable named *darctemp* to store the value of dummy node's arcs value. While *nodejarc* is used for store the value of a node (father node) 's arcs value. It is clear from Figure 4.10 that we leave a space gap for each dummy node and therefore allow two nodes away from more than 2 depths can 'reach to' each other. Actually, due to the right formulation of this idea, as long as the original graph shown in left side is a planar st-graph, we can in the right side draw the nodes by the orders where they should be in original graph. However, the only thing here may generate bugs is the size of dummy node, i.e., the value of *darctemp*. The simplest way to do that is to assign *darctemp* a small and fixed value, e.g.  $\pi/12$ , but this is not a correct way. For instance in Figure 4.10, if the arc value of **node 2** is smaller than  $\pi/12$ , then the whole structure will be damaged. So here we will assign the value to *darctemp* dynamically:

```

    if (nodejarc > 0.5)
darctemp = (float) (nodejarc / 10);
else
darctemp = (float) (nodejarc / 3);

```



The reason to do it this way is because if the father node which dummy node belongs to is too small or too big comparing with other nodes, then one fixed percentage will make the gap (value of darctemp) too big or small.

## Algorithms

Begin by checking the second layer (**Layer 1**) firstly:

### STEP1

```
1: for every node in Layer 1:
2: if (node == LayerNode.Node) Nnum++;
3: if (node == LayerNode.DummyNode) Dnum++;
4: nodejarc ← node.arcs(node.stop-node.start)
5: Assign value to darctemp;
6: arctemp ← (nodejarc-Dnum*dractemp)/Nnum;
```

### STEP2

```
1: for every node in nodej's LayerSet:
2: for the first node
   check 'start' switch
if(true)
   thisnode.start ← nodej.start; switch off
3: check 'stop' switch
if(true)
   thisnode.stop ← thisnode.start + thisnode.arc value (this value differ from node and dumminode ); switch off
```

### STEP3

*for the rest nodes*

```
1: check switch
if(true)
thisnode.start ← prenode.stop;
thisnode.stop ← thisnode.start + thisnode.arc value (this value differ from node and dumminode );
end for
end for
```

### STEP4

*loop from layer 2 to the layer (layer.length - 1)*

When all this jobs have been done, the final step is to draw all the arcs on screen with a for loop of a set consists of all the LayerNodes' information, in this case is the set called "output". Only arcs is not sufficient, we also need to draw

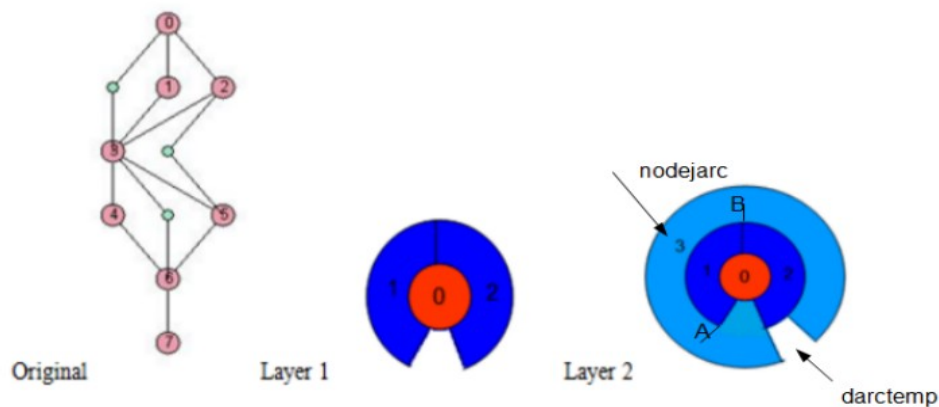


Figure 4.10: assign space for dummy node

the lines in order to distinguish the nodes in the same layer. And here we adopt the  $\cos()$  and  $\sin()$  function to calculate the parameters for two lines (from centre to start, from centre to stop) for each arc in order to conduct the sector.

```
(cos((float)x.getstart()),x.GetDeep()*templine*(sin((float)x.getstart())));
(cos((float)x.getstop()),x.GetDeep()*templine*(sin((float)x.getstop())));
```

This part of code is used for drawing the sectors. When the graph is drawn on screen, we also have two kinds of interactions as discussed in Section 3.6, brushing and tooltib and all of these two functions will be implemented inside *mouseMoved()* with *Processing* API.

```
public void mouseMoved()
{int id = picker.get(mouseX, mouseY);
fill(255);
rect(mouseX-175,mouseY-250,50,25);
fill(0);
if (id==-1)
text('','','',mouseX-170,mouseY-234);
else
text('Node:''+id,mouseX-170,mouseY-234);
output = analyzer.GetList().GetSet();
_id = id; }
```

The tooltib function can be implemented in this function, and the brushing function is implemented in the *draw()* function of *DrawArc* class in which need to use the id from this function. The output of the final result will be shown in Figure 4.11.

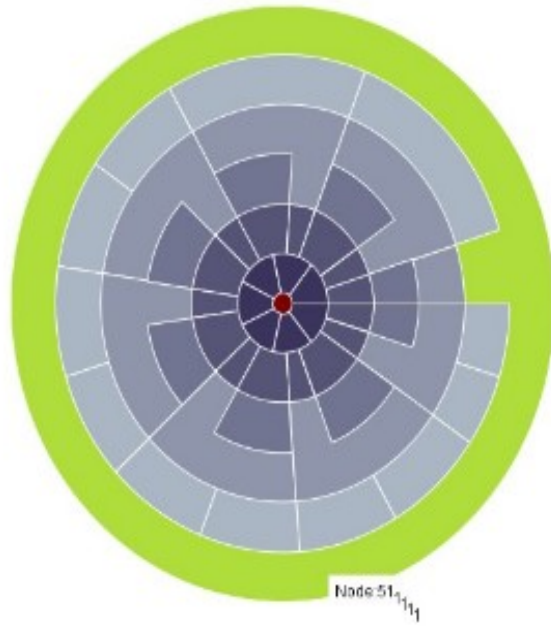
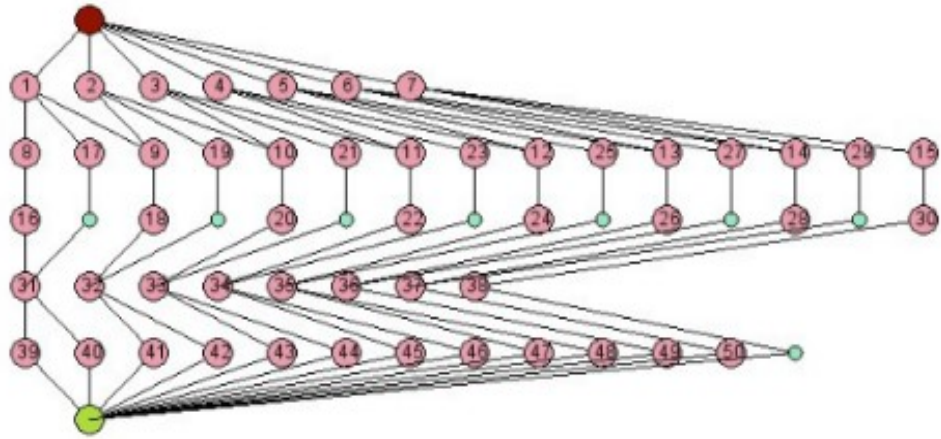


Figure 4.11: The final output

## 5 Discussion

By providing a new idea of representation with space-filling technique, the final results of this project overcome the problems for traditional layout of planar st-graph that waste of space and difficult to distinguish relations between edges and nodes. Taking Figure 4.11 as example: This is a specific planar st-graph with 52 nodes including both source and terminal nodes. In this figure, traditional representation of the graph is shown at the top, in which there are three features that can be identified with first sight: a) the number of layers, b) the number of nodes and c) whole structure of the graph. However, a lot of space are wasted on this graph which include both vertical and horizontal dimensions. For example, the gap between layer 2 and 3, or the space between node 8 and node 17.

From Figure 4.11, the reason of why space are wasted in traditional approach of planar st-graph representation is because: the connections between nodes are represented by edges, which implies the increasing utilization of vertical space; moreover, gap between nodes in the same level also need to be assigned with a proper space in order to ensure the right distinctions among nodes and their connected edges. For example, it is hard to follow the path from node 4 to node 45 although there indeed has a path. This may due to the deficient representing approach for traditional planar st-graph drawing in this project, however, even given by a better drawing algorithm, it still has the problem of identifying nodes and edges when too many of them exist in a little space at the same time.

The bottom of Figure 4.11 is another approach for representing the same planar st-graph but with new space-filling techniques that developed in this project. In this case, each curve unit represents the corresponding node and space are maximum utilized because the relations between nodes are represented by the adjacent relations of curve units. Every curve unit has two kinds of curving edges: top and bottom, of which two curve units are connected when and only when one unit's top curving edge 'adjoin' another unit's bottom curving edge. We can see from the figure, there is no white space in this graph representation excepting the white lines used for distinguishing each curve unit (node). Moreover, by hierarchical color-coding, people can intuitively distinguish between two contiguous layers and identify the connections between nodes from different layers. In addition, the new approach of planar st-graph visualization can better support user interactions due to its ductility of the representing structure and the more available space.

What is more, this project can give people who want to improve the graph drawing area a start point to think about how to make the layout more intuitive so that it can better fit human's perspectives, and to think about how to save the space on screen and how to fulfill the requirement of presenting the overview and context.

However, as we said, this is only a start point of the whole idea. There are two main limitations in this project. a) On the original graph aspect, we did the

edge crossing reduction manually which will cost a lot time when running the program and it requires the user do it properly in order to come to the right layout and structure of the new visualization. b) On the new graph side, we have less interactions, this can also generate limitations when a node has too many children nodes – the space for each child node will become too small to distinguish.

## 6 Future Work

As mentioned before, there are some limitations exist in this project, thus here we will list the corresponding future works on it trying to solve the problems and improve the quality.

```
procedure Random-First-Improvement-Iterative-Local-Search
  input: problem instance  $\pi$  (ie. a complete rectilinear graph on  $n$  vertices),
  objective function  $F$ , optimal crossings  $OPT$ 
  output: solution  $s$  (ie. a complete rectilinear graph on  $n$  vertices with optimal number of edge crossings)
   $s := initialize(\pi)$ ;
   $current = F(s)$ ;
   $numTries := 0$ ;
  while  $current > OPT$  and  $numTries \leq maxTries$  do
     $s' := move-random-vertex(s)$ ;
     $temp = F(s')$ ;
    if ( $temp \leq current$ )
       $s := s'$ ;
       $current = temp$ ;
       $numTries++$ ;
  end while
  if ( $degeneratePoints(s) == false$  and  $collinearity(s) == false$ )
    return  $s$ ;
  else
    errorMessage();
end procedure Random-First-Improvement-Iterative-Local-Search.
```

Figure 6.1: RFI algorithm

### *1 Eliminating edge crossing automatically*

Actually, in computer graph drawing area, there are a lot of algorithms to eliminate edge crossing. For example, Maxwell, Y. et al [19] had developed several kinds of minimizing edge crossing algorithms one of which named "Random First improvement Iterative Local Search" (RFI) as shown in Figure 6.1. By implementing this kind of algorithm, the limitation a) will be overcome automatically.

### *2 Additional interactions on this project*

One good interaction is to adopt sunburst [16] technique to focus on one specific node and to see how other nodes related to this one. As Figure 2.8 shows up, when we click *radiation* node, other nodes which have no connections will become gray and therefore we can highlight the proper range that we may wish to explore. What is more, we can also make the *radiation* node become big when we click on it so that the limitation of too many children nodes in one LayerSet will



Figure 6.2: Motivation of interactions [16]

be improved.

## References

- [1] Diestel, Reinhard (2005). *Graph Theory* (3rd ed.). Berlin, New York: Springer-Verlag.
- [2] Michael T.G. & Stephen G.K. , 2002, *Graph drawing:10th international symposium*, GD 2002, Irvine, CA, USA, August 26-28, 2002 : revised papers Berlin : Springer, cop. 2002
- [3] Tube - map. London. Last access: July 15 2011, from <http://www.afn.org/alplatt/tube.html>
- [4] Tollis, I.G., et al, 1998, *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall; 1 edition
- [5] McKay, Brendan; Brinkmann, Gunnar, *A useful planar graph generator*.
- [6] Mark B. Kees H and Jarke J, *Squarified Treemaps*. Eindhoven University of Technology Dept. of Mathematics and Computer Science.
- [7] I.G. Tollis & C. Papamanthou, 2010, Applications of Parameterized st-Orientations in Graph Drawing Algorithms, *Journal of Graph Algorithms and Applications*. Vol.14, No.2, pp.337-365
- [8] Keith A., 2012, *Information Visualization*. IICM Graz University of Technology
- [9] Atallah, M.J., et al, 1998. Efficient Parallel Algorithms For Planar st-Graph. *Computer Science Technical Reports*. Paper 1410.
- [10] Di B.G., et al, 1989, Automatic layout of PERT diagrams with X-PERT, *Visual Languages*.
- [11] Boyer, John M.; Myrvold, Wendy J. (2005), "On the cutting edge: Simplified  $O(n)$  planarity by edge addition", *Journal of Graph Algorithms and Applications* 8 (3): 241-273.
- [12] Mohamed A. El-Sayed, 2012. GA for straight-line grid drawings of maximal planar graphs. *Egyptian Informatics Journal* 13(1)
- [13] T. Nishizeki et al, 2008. *Planar Graphs: Theory and Algorithms*. Dover Publications
- [14] Brian, J. and Ben, S., 1991. Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures. *Proceedings IEEE Visualization (1991)* .
- [15] Tyson R. Henry and Scott E. Hudson. Viewing large graphs. *Technical Report 90-13*. University of Arizona, May 1990.



- [16] Hans, S., 1994. *Space-Filling Curves*. Springer; 1 edition
- [17] F. Ben & R. Casey, 2007. *Processing : a programming handbook for visual designers and artists*, Cambridge, Mass. : MIT Press
- [18] GraphML Primer. Last access: May 20 2011, from <http://graphml.graphdrawing.org/primer/graphmlprimer.html>
- [19] Maxwell, Y. et al, 2003. Stochastic Local Search Algorithms for Minimizing Edge Crossings in Complete Rectilinear Graphs.



# **Linnæus University**

School of Computer Science, Physics and Mathematics

SE-391 82 Kalmar / SE-351 95 Växjö

Tel +46 (0)772-28 80 00

[dfm@lnu.se](mailto:dfm@lnu.se)

[Lnu.se/dfm](http://Lnu.se/dfm)