NOTE

This online version of the thesis may have different page formatting and pagination from the paper copy held in the Swinburne Library.
Filtering, Clustering and Dynamic Layout for Graph Visualization

A Dissertation submitted to
The School of Information Technology
Swinburne University of Technology

by

Xiaodi Huang
B.Sc. (Hons.), M. Phil

For the award of

DOCTOR OF PHILOSOPHY

2004
Acknowledgements

I would like to express my deep gratitude to my supervisor, Dr Wei Lai, for providing me with supervision, inspiration and motivation throughout the course of this work. I would also like to express my thanks to my co-supervisor, Associate Professor Doug Grant.

I would like to express my sincere appreciation to the School of Information Technology, Swinburne University of Technology, for offering me a research scholarship throughout my doctoral program, and for providing me with the financial support to attend conferences.

My thanks also go to other staff members of the School, and to Mr Ronald Hendra Wibowo, Mr Haizheng Dong, and Mr Gangdong Yu.

I am deeply grateful to my wife Weidong Zheng for her love, patience, understanding and encouragement.
Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma, except where due reference is made in the text of the thesis. To the best of my knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Signed

Dated 10/12/2003
To my wife Weidong Zheng, my daughter Rui Huang,
and my son Alan Huang
Abstract

Graph visualization plays an increasingly important role in software engineering and information systems. Examples include UML, E-R diagrams, database structures, visual programming, web visualization, network protocols, molecular structures, genome diagrams, and social structures.

Many classical algorithms for graph visualization have already been developed over the past decades. However, these algorithms face difficulties in practice, such as the overlapping nodes, large graph layout, and dynamic graph layout. In order to solve these problems, this research aims to systematically address both algorithmic and approach issues related to a novel framework that describes the process of graph visualization applications. At the same time, all the proposed algorithms and approaches can be applied to other situations as well.

First of all, a framework for graph visualization is described, along with a generic approach to the graphical representation of a relational information source. As the important parts of this framework, two main approaches, Filtering and Clustering, are then particularly investigated to deal with large graph layouts effectively.

In order to filter “noise” or less important nodes in a given graph, two new methods are proposed to compute importance scores of nodes called NodeRank, and then to control the appearances of nodes in a layout by ranking them.

Two novel algorithms for clustering graphs, KNN and SKM, are developed to reduce visual complexity. Identifying seed nodes as initial members of clusters, both algorithms make use of either the $k$-nearest neighbour search or a novel node similarity matrix to seek groups of nodes with most affinities or similarities among them. Such groups of relatively highly connected nodes are then replaced with abstract nodes to form a coarse graph with reduced dimensions.

An approach called MMD to the layout of clustered graphs is provided using a multiple-window–multiple-level display.
As for the dynamic graph layout, a new approach to removing overlapping nodes called Force-Transfer algorithm is developed to greatly improve the classical Force-Scan algorithm.

Demonstrating the performance of the proposed algorithms and approaches, the framework has been implemented in a prototype called PGD. A number of experiments as well as a case study have been carried out.
## Contents

**Chapter 1  Introduction**.............................................................................................................. 1  
1.1 Motivation.............................................................................................................................. 1  
1.2 Objectives ............................................................................................................................ 2  
1.3 Contributions ........................................................................................................................ 3  
1.4 Organization.......................................................................................................................... 5  

**Chapter 2  Literature Review**...................................................................................................... 7  
2.1 Graph Layout .......................................................................................................................... 7  
2.1.1 Drawing Conventions ......................................................................................................... 8  
2.1.2 Aesthetic Criteria .............................................................................................................. 9  
2.1.3 Drawing constraints .......................................................................................................... 11  
2.1.3 Graph Drawing ................................................................................................................ 12  
2.2 Classifications of Graph Layout .......................................................................................... 13  
2.3 Graph Layout Algorithms..................................................................................................... 14  
2.4 Interaction and Large Techniques........................................................................................ 20  
2.4.1 Large Graph Layout ........................................................................................................ 20  
2.4.2 Dynamic Graph Layout .................................................................................................. 26  
2.3 Summary............................................................................................................................... 27  

**Chapter 3  A Framework of Graph Visualization**................................................................. 28  
3.1 Introduction............................................................................................................................ 28  
3.2 Related Work ....................................................................................................................... 29  
3.2.1 A Task by Data Type Taxonomy for Information Visualization................................. 29  
3.2.2 Information Visualization Data State Reference Model............................................. 30
Chapter 4  NodeRank: Filtering for Layout

4.1  Introduction ................................................................. 51
4.2  NodeRank: Node Importance Rank ........................................ 53
  4.2.1  Degree, Closeness and Between Centrality Index .............. 54
  4.2.2  Eigenvector Centrality ..................................................... 57
4.3  Cutsets and Bridges .......................................................... 59
4.4  Filtering Graphs ................................................................. 62
  4.4.1  Global Filtering .............................................................. 62
  4.4.2  Fisheye View ................................................................. 63
4.5  Algorithms ........................................................................ 64
4.6  Filtering Examples .............................................................. 65
  4.6.1  A Small Documents Collection on “receipts” .................... 66
  4.6.2  Web Graphs ................................................................. 69
4.7  Summary ........................................................................ 78

Chapter 5  KNN: Clustering for Layout ........................................ 80

5.1  Introduction .................................................................. 80
Chapter 6  SMK: Clustering for Layout ................................................................. 99

6.1  Introduction ................................................................................................. 99
6.2  Node Vector Space Model ............................................................................. 100
6.3  Node Similarity Matrix .................................................................................. 101
6.4  Definitions .................................................................................................. 105
6.5  Algorithms .................................................................................................. 108
   6.5.1  Determination of the Number of Clusters and Initial Nodes .............. 108
   6.5.2  K-means ............................................................................................... 109
   6.5.3  Main Algorithm .................................................................................... 111
6.6  Examples ...................................................................................................... 112
6.7  Comparison KNN with SMK ....................................................................... 119
6.8  Related Work .............................................................................................. 120
6.9  Summary ..................................................................................................... 121

Chapter 7  MMD: Layout of Clustered Graphs .................................................... 123

7.1  Related Work ................................................................................................ 123
7.2  Multilevel Multi-Window Drawings ............................................................. 124
7.3  Algorithms ................................................................................................... 131
   7.3.1  Window Expansion/Shrink ..................................................................... 131
   7.3.2  The Modified Spring Algorithm ............................................................ 133
   7.3.3  Dynamic Adjustment and Removing Overlapping Nodes or Windows 134
7.4  Summary ...................................................................................................... 134
List of Figures

Figure 2.1: Example of Drawing Conventions ................................................................. 8
Figure 2.2: Three Dimensional Classification of Graph Layout ................................. 13
Figure 2.3: Spring Embedder Algorithm ........................................................................ 17
Figure 2.4: The Process of Coarse and Fine Graphs [From Yehuda Koren et al.] .... 21

Figure 3.1: Information Visualization Data State Reference Model [Riedl 1998] .... 30
Figure 3.2: Architecture for Visualization of Large Graph [Peter Eades 2000] .... 31
Figure 3.3: A Framework of Graph Visualization Pipeline ........................................ 35
Figure 3.4: Example of Cosine Threshold ..................................................................... 42
Figure 3.5: The Transformation Relationships during Graph Modeling .................... 44
Figure 3.6: An Interface of the PGD Framework ......................................................... 47
Figure 3.7: Another Interface of the PGD ................................................................. 48

Figure 4.1: Rule-Based Filtering (From Huang et al.) ................................................ 53
Figure 4.2: An Example of Cutpoint and Bridge ......................................................... 60
Figure 4.3: A Random Graph and its Filtered Graph ................................................ 66
Figure 4.4: The Structure of A Small Document Collection on Recipes .................. 67
Figure 4.5: Comparison of Centrality Indices and Node Importance Scores in Table 4.1 ........................................................................................................................................ 68
Figure 4.6: Comparisons of Node Importance Values by DCB and Eigenvector in Table 4.2 ........................................................................................................................................ 70
Figure 4.7: A Web Graph ................................................................................................ 74
Figure 4.8: Part of Web Graph of Web Site of Swinburne University of Technology ........ 76
Figure 4.9: Part of Web Graph of Web Site of the University of Melbourne ............. 78

Figure 5.1: An Algorithm to Find Seed Nodes ........................................................ 88
Figure 5.2: Algorithms to Aggregate Nodes by Local Search .................................. 89
Figure 8.5: An Example of An Overlapping Graph Adjusted by FTA And FSA .... 154
Figure 8.6: Initial and Adjustment Layouts for Graph 1 .............................. 155
Figure 8.7: Initial and Adjustment Layouts for Graph 2 .............................. 155
Figure 8.8: Initial and Adjustment Layouts for Graph 3 .............................. 156
Figure 8.9: Initial and Adjustment Layouts for Graph 4 .............................. 156
Figure 8.10: Initial and Adjustment Layouts for Graph 5 ........................... 156
Figure 8.11: Initial and Adjustment Layouts for Graph 6 ............................ 157
Figure 8.12: Initial and Adjustment Layouts for Graph 7 ............................ 157
Figure 8.13: Initial and Adjustment Layouts for Graph 8 ............................ 158
Figure 8.14: Initial and Adjustment Layouts for Graph 9 ............................ 158
Figure 8.15: Comparison of Values of Cost Function for Graph 3-Graph 9 .... 159

Figure 9.1: Hypergraph ............................................................................... 165
Figure 9.2: Navigation of a Web Site ......................................................... 168
Figure 9.3: Process of an Example of Visual Web Navigation ...................... 169
Figure 9.4: An Example of Filtering ........................................................... 170
Figure 9.5: A Clustered Web Graph ............................................................ 171
Figure 9.6: Vertical Exploration of the Web Site ........................................ 173
Figure 9.7: Horizontal Exploration of the Web Site .................................... 174
Figure 9.8: Dynamic Adjustment of the Web Site ....................................... 176
List of Tables

Table 4.1: The Centrality Indices of Documents in the Collection .........................68
Table 4.2: The Centrality Indices of Nodes in A Web Graph of Figure 4.7 ...............71

Table 5.1: The Parameters for Clustering Graphs 1-5 ...........................................94

Table 6.1: The Node Similarity Matrix of the Graph in Figure 6.1 .........................104

Table 8.1: Value of Cost Function $F(V_i)$ ..............................................................159
Chapter 1

Introduction

1.1 Motivation

As a fundamental structure in computer science, graphs are commonly employed to model relational structures that are sets of entities or objects and the relationships between them. In other words, graphs provide a very natural notation for many areas in computer science. Graph layout is a conventional tool for visualization of such relational information where the objects are represented as nodes, and the relations between these objects as edges. Since graphs are ubiquitous models, graph structures are among the most commonly used types of visual representations. For this reason, the automatic layout of graphs plays an important role in a wide range of applications:

- Visual Programming
- Software Engineering
  UML, data flow charts, state charts, dependency visualization, schemas, E-R diagrams, database structures, repository structures, and so on.
- Network
  Web visualization, Web access patterns, Petri nets, network protocols, telephone systems, and so on.
- Engineering
  Circuit diagrams, molecular structures, chemical formulae, genome diagrams and biochemical pathways.
- Social science
  Genealogy, social structures, and so on.

The aim of producing a graphical representation of relational information is to make the information easier to understand. In particular, the graphical representation is extremely useful in illustrating complex structures in a direct and intuitive way. The fundamental issue in graph drawing is to develop graph drawing algorithms that take a
CHAPTER 1. INTRODUCTION

graph and produce a good drawing for it. More accurately, a drawing is a function that associates an output area (drawing window) with a graph.

In recent years, many algorithms for drawing graphs automatically have been proposed [Alpert et al. 1995; Battista et al. 1994; Battista et al. 1999; Eades et al. 1990; Herman et al. 2000]. There are also many graph drawing systems, such as Graphlet [Himsolt 2000; Himsolt 1996], Graph Layout Toolkit [Dogrusöz et al. 1996], and others [Emden et al. 2000; Giuseppe et al. 2002; Huang and Eades 1998; Riedl 1998; Rodolfo 2002]. These systems provide basic toolkits for graph editing and management, and implement graph drawing algorithms.

Although there is a wealthy body of graph drawing algorithms (Chapter 2), and many graph drawing systems are available, the systematic development of algorithms with respect to a framework of graph drawing applications has received little attention. Our research is motivated by an attempt to address algorithmic and approach issues that are related to a framework describing the process of graph drawing applications. At the same time, all the proposed algorithms and approaches are also generically applicable to other situations.

1.2 Objectives

This research attempts to investigate algorithms and approaches associated with a proposed framework. This framework is, in turn, built on the basis of the analysis of graph drawing applications. In particular, the objects of this research are:

- To propose a framework that relies on the stages of graph drawing applications.
- To develop algorithms and approaches related to the above framework.
- To conduct case studies by applying the proposed algorithms and approaches.

We will examine frameworks of graph drawing applications, and then develop effective algorithms and approaches to implement the proposed framework. In addition, our research will focus mainly on the technical rather than the design issues.
1.3 Contributions

The contributions of this thesis fall into three areas: a framework, algorithms and approaches.

Our framework contribution is a novel framework, which describes the stages of a graph drawing application. Different algorithms and approaches are developed in each stage to achieve various views of a drawing by users.

Our algorithmic contributions include several novel algorithms for graph modeling, filtering, clustering, and dynamic layout.

Our proposed approach is an improved one for the layout of clustered graphs.

These contributions are detailed as follows:

1. PGD: A new framework for practical graph layout is proposed, together with a general approach to a graphical representation of given relational information. Based on the given premises, the approach used at the first stage of the proposed framework is particularly developed for automatic extraction of a graph representation from an information source.

2. NodeRank: We present two methods to compute NodeRank whose value ranges from 0 to 1, measuring the degree of importance the role of a node plays in a graph. These methods are derived from the methodology in social network analysis. Moreover, an algorithm for graph filtering is presented on the basis of NodeRank. Two models of filtering a graph are described, namely Global Filtering and Fisheye. Our approach makes use of the structural features of a graph so that it is flexible and has wide applications compared with the rule-based approach. Graph filtering is an alternative way to deal with large graphs.

3. KNN and SMK: Two novel algorithms of graph clustering for the layout. Starting with the identification of seed nodes, special nodes in a graph, both algorithms group those nodes that have high affinities with, or most similar to, existing member nodes, into clusters. The advantages of the two algorithms are that they are specially developed for graph layout. They are also applicable to clustering an arbitrary graph.

KNN and SMK have been published in the follow proceedings [Huang and Lai. 2003], respectively:


4. FTA: A novel algorithm for the dynamic graph layout and removing overlapping nodes, called Force-Transfer Algorithm (FTA), is developed on the basis of the Force-Scan algorithm. We validate our algorithm by providing both theoretical proof and experimental examples. In terms of compact adjustments and time complexity, our approach is a good candidate for the adjustment of an overlapping graph layout, in comparison with other approaches.

FTA has been published in [Huang and Lai 2003]:


5. MMD: A new approach to the layout of clustered graphs.
MMD has been published in [Huang and Lai 2003]:


6. A prototype of the proposed framework.

### 1.4 Organization

This thesis is organized to highlight several research problems in graph layout, which come from the proposed framework.

This thesis begins with our motivation for graph drawings and the objects of this research, followed by a summary of our original research contributions. We provide a literature review in Chapter 2.

A framework of graph layout applications is presented in Chapter 3. In addition, we describe a generic approach to constructing a graph on the premise that a set of object attributes are given.

Chapter 4 presents a method for filtering graphs based on *NodeRank*. We provide two approaches to calculating *NodeRank*, and then examine two filtering models, namely global filtering and Fisheye, with several illustrative examples.

The next two chapters, Chapters 5 and 6, present two algorithms for clustering graphs.

Chapter 5 addresses an approach to clustering a graph for the layout. Starting with formalizing the quality of a clustered graph, an algorithm called KNN will be presented.
CHAPTER 1. INTRODUCTION

Chapter 6 proposes another algorithm for clustering graphs. We construct a similarity matrix to reflect the linkage pattern of a graph, and then group the most similar nodes into a cluster by the $K$-means algorithm with some given input.

Chapter 7 is about a multi-level multi-window layout of a clustered graph, called MMD, and illustrative examples are provided.

For dynamic graph layout and the problem of removing overlapping nodes, we develop a novel algorithm called Force-Transfer Algorithm (FTA) presented in Chapter 8, greatly improving the Force-Scan Algorithm (FSA). Both theoretical proofs and empirical examples are presented.

Chapter 9 acts as a case study, applying the proposed algorithms and approaches to the visualization of a Web site.

We finish this thesis with discussions, future work, and conclusions in Chapter 10.
Chapter 2

Literature Review

This chapter presents several basic terminologies and concepts in graph layout. A classification of graph layout is illustrated. We then describe the basic algorithms that are widely referred to in the literature of graph layout, and review several recently developed algorithms, particularly for the large and interactive graph layout. Finally, we briefly discuss the limitations of current graph layout algorithms.

2.1 Graph Layout

A graph, a fundamental structure in computer science, is an abstract mathematical object [Berge 1993; West 1996]. The following is quoted from “Mathworld”\(^1\)

“In a mathematician's terminology, a graph is a collection of points and lines connecting some (possibly empty) subset of them. The points of a graph are most commonly known as graph nodes, but may also be called "nodes" or simply "points". Similarly, the lines connecting the nodes of a graph are most commonly known as graph edges, but may also be called "arcs" or "lines".

We start with the definition of a graph and then discuss some premilaries of graph layout.

Definition 2.1 Graph: An abstraction of the entities (or objects) and their relationships in an information space, defined as \( G = (V, E) \), where \( V \) is a set of entities (or objects), and \( E \) is the set of relationships between the entities.

There are also several sub-categories of graphs. If all relations are between two objects, the graph is an ordinary graph; otherwise it is a hypergraph. If the same set of

\(^1\) http://mathworld.wolfram.com/
objects is related in several different ways, the graph will have multiple or parallel edges called a multigraph.

2.1.1 Drawing Conventions

The definition of drawing conventions is provided in this section, along with an illustrative example.

**Definition 2.2 Drawing Conventions:** The basic drawing styles for the representation of graphs.

Several most important conventions are listed below.

- **Grid drawing:** Each node has integer coordinates (Figure 2.1).
- **Polyline drawing.** Each edge is a polyline, or a chain of line segments. This means that the final drawings can have as many bends in the edges as desired.
- **Straight-line drawing.** Each polyline in a polyline drawing is a line segment. In other words, each edge is a straight-line segment so that there are no bends.
• **Orthogonal drawing:** Each edge is represented by a polyline composed of straight line segments that are parallel to a coordinate axis. Each edge in an orthogonal drawing is a chain of horizontal or vertical segments.

Other drawing conventions include planar drawings that minimize crossings and bends.

### 2.1.2 Aesthetic Criteria

The primary requirement of graph layout algorithms is that the output layout should be readable (easy to understand and follow), and should represent a set of edges and nodes in such a way that will convey the maximum meaning. We should thus clarify the following questions: what makes drawings understandable or readable? What is a nice drawing? Is it possible that one graph layout can be objectively better than others in terms of conveying information? The key point to answering these questions is aesthetic criteria for layouts, which refers to the attributes of a good graph layout.

**Definition 2.3 Drawing Aesthetic Criteria:** The common rules for improving the readability of a drawing.

The qualities affecting the readability of a graph are listed:

- Minimize crossing – keep the number of times that lines cross to a minimum.
- Minimize area – keep the area that the graph takes up to a minimum by producing a compact graph.
- Minimize the sum of the edge lengths.
- Obtain a uniform edge length – try to keep each of the edges the same length.
- Minimize bends – keep the number of bends to a minimum.

From another perspective, user studies [Batini 85; Cohen 94] claim that a drawing is more readable if it satisfies the following readability criteria:

- has no edge-bends
- has no edge-crossings
- has small edge-lengths
CHAPTER 2. LITERATURE REVIEW

- occupies a small screen-space
- displays symmetries
- satisfies application-specific constraints such as clustering and alignment

How can we quantitatively measure the quality of a graph drawing, and can we formalize aesthetics? Helen C. Purchase [Purchase 2002] presented formal metrics for measuring the above common aesthetic criteria that specially measure the extent to which a graph drawing conforms to each of seven common aesthetic criteria, by a real value ranging from 0 to 1.

Suppose a graph drawing \( D(G) \) that has \( n \) nodes and \( m \) polyline edges, and a derived \( D'(G) \) with \( n' \) nodes and \( m' \) straight-line edges. Some common metrics are then be formalized:

- Crossings
  \[
  \zeta_c = \begin{cases} 
  1 - \frac{c}{2} & \text{if } \frac{m'(m'-1)}{2} - \frac{1}{2} \sum_{i=1}^{n'} \deg(v_i)(\deg(v_i) - 1) > 0 \\
  0 & \text{otherwise}
  \end{cases}
  \]
  where \( c \) is the number of edge crossings in \( D(G) \), and \( \deg(v_i) \) is the degree of node \( v_i \).

- Bends
  \[
  \zeta_b = 1 - \frac{m' - m}{m'}
  \]

- Minimum Angle
  \[
  \zeta_m = 1 - \frac{\sum_{i=1}^{n'} \left| \frac{360^\circ}{\deg(v_i)} - \vartheta_i^{\text{min}} \right|}{360^\circ}
  \]
  where \( \vartheta_i^{\text{min}} \) is the actual minimum angle between the incident edges at node \( v_i \).

For the other metrics, such as Symmetry, Orthogonality and Upward flow, refer to the above paper [Purchase 2002].

Graph drawing can also be viewed as an optimization problem. Various cost functions are therefore defined to measure the aesthetic criteria.
Recently, specific aesthetics were proposed for the drawings of large graphs [Harel 2000]. The basic rule is that the drawings should be “nice” for the coarse graphs, including global and local levels. In particular, global aesthetics indicate the requirement of preserving a nice global structure of the original drawing. In contrast, local aesthetics refer to the requirement of smaller areas of the drawings.

Although there are several kinds of drawing aesthetic criteria, the purpose of all the criteria is to construct readable drawings as indicated before. In real applications, the selection of those criteria depends on the types of applications.

### 2.1.3 Drawing constraints

Drawing constraints differ from *Drawing Aesthetics Criteria* in that they vary among applications.

**Definition 2.4 Drawing Constraints:** Geometric relationships placed on a drawing that should be satisfied by a drawing algorithm.

Applying constraints to graph drawings has two purposes. The first is that users are able to pre-arrange certain shapes, and second, allow more control of the outcome of the drawing. Also, with constraints, the process of graph drawings can be greatly speeded up. The constraints that can be commonly placed on a graph drawing algorithm are as follows:

- Center – place a given node in the center
- External – place a given node on the boundary
- Cluster – place a group node together
- Pre-arrange a path
- Draw a subgraph with a certain shape

Drawings satisfying these constraints sometimes cannot be found, for example, in the presence of conflicting constraints. Complex interactions between many constraints may lead to such a problem, and therefore can be difficult to achieve.
CHAPTER 2. LITERATURE REVIEW

2.1.3 Graph Drawing

Technically, the central objective of a graph drawing algorithm is to assign a location to each node and a route to each edge, so that the resulting drawing is "nice" and clearly conveys the relational structure.

We are concerned here with the problem of drawing an undirected graph with straight-line edges.

As mentioned above, the basic graph drawing problem can be alternatively described as: given a set of nodes with a set of edges (relations), calculate the positions of the nodes and the curve to be drawn for each edge.

Definition 2.5 Graph Layout (or Drawing)²: A field of relational visualization that concerns itself with assigning positions in the visualization space to entities and relationships from an information space. Formally, given a graph $G = (V, E)$ and drawing conventions, an algorithm for graph layout finds a map $f : G \rightarrow R^n, n = 2,3$ (2D or 3D Cartesian space), subject to the aesthetic criteria and (or) drawing constraints defined previously.

In other words, graph layout is a transformation from topology to geometry which assigns coordinates to the nodes (placement of nodes) and the routing of edges, with the purpose of finding an aesthetic layout of a graph that clearly conveys its structure.

Graph layout problems can also be treated as a particular class of combinatorial optimization problems whose goal is to find a layout of an input graph in such a way that a certain objective function is optimized.

¹ In this thesis, the terms graph layout and graph drawing are used interchangeably.
2.2 Classifications of Graph Layout

There are a number of well-known techniques for graph layout, such as Spring Embedder, Sugiyama algorithm and so on. A good overview of the algorithms can be found in [Battista et al. 1994; Battista et al. 1999; Alpert and Kahng 1995; Herman et al. 2000; Diaz 2002; Kaufmann and Wagner 2001]. The fundamental questions of graph layout are: What kind of a graph is to be laid out. Which algorithm can be effectively applied? In addition, how does an algorithm deal with a large graph and a dynamic layout? The available techniques for graph layout can thus be classified on the basis of these three questions.

![Three dimensional classification of graph layout](image)

According to Shneiderman’s taxonomy of data types [Shneiderman 1996], there are several data types: 1D and temporal data, 2D (maps, scatterplots), 3D (e.g. the world), hierarchical (e.g. trees), graph data (e.g. Web pages) and links (multidimensional data). Different data types utilize various visualization techniques: 1D Data: scrolling list, scrollbars; 2D: Perspective Wall [Mackinlay 1991], ‘fisheye’ lens [Formella and Keller 1995; Furnas 1986; Furnas 1981; Nagumo 2001; Sarkar and Brown 1994; Schaffer et al. 1996], scatterplots and maps [Wills 1998]; 3D: the ‘natural’ world; Hierarchical Data, Cone trees [Carrière and Kazman 1995; Robertson et al. 1991], Cam trees [Cordella et al. 2001].
The commonly used types of graphs for the layout are as follows: trees (ordered trees), hierarchical (radial, embeddings on the grid, and other techniques), acyclic graphs, directed acyclic graphs (Sugiyama algorithm), general graphs (force-directed approaches, and multi-dimensional approach), planar graphs (straight line, orthogonal, Tamassia’s flow technique and visibility), and others (two stage approaches).

The fundamental algorithms in graph layout include force-directed methods for general graphs, Sugiyama algorithm for directed acyclic graphs, Reingold-Tilford for trees, and Tutte embeddings for planar graphs. The major algorithms of graph layout will be described in detail in the next section.

The third dimension of the classification is the Interaction and Large Graph Layout Techniques, which allow users to directly interact with a layout to get different insights, especially for a large graph. The techniques fall into the following categories: general interaction techniques (filtering, categorization, multiple views and so on), interaction techniques (varied views, abstractions and interpretations), and user control and adaptive techniques (filtering, categorization and clustering, zooming and semantic zooming, getting hidden detail and multiple views).

### 2.3 Graph Layout Algorithms

There are many graph layout algorithms available in the literature. In the following we review only two fundamental techniques for graph layout.

**Force-Directed Methods for General Graphs**

The general idea of the force-directed method is to work on the physical model of a graph in which the nodes represent steel rings and the edges indicate springs attached to these rings (Figure 2.3). In other words, a force model is associated with a graph.

Three kinds of forces are considered in various force models:
CHAPTER 2. LITERATURE REVIEW

- Attractive force

\[ f_a(u, v) = \frac{\| (u, v) \|^p}{k} \]

The attractive forces are along each edge and are proportional to the shortest paths between two corresponding nodes.

- Repulsive force

\[ f_r(u, v) = \frac{k^p}{\| (u, v) \|} \]

The repulsive forces are between each pair of nodes or only between closely related nodes.

The parameters in the above two equations are:

- \( k \): an ideal edge length calculated by \( k = 0.75 \cdot \frac{\text{area}}{\sqrt{|V|}} \), where the area is the size of a display window.
- \( p \): the value of 2 or 3.
- \( u \) and \( v \): two nodes in a graph.

In addition, the above forces can be linear, logarithmic [Eades 1984], quadratic [Fruchterman and Reingold 1991], and cubic [Forster et al. 1999].

- Other forces

Apart from the above two kinds of forces, other forces include the following: centre of gravity (attractive), underlying magnetic fields (concentric, radial, and horizontal) [Sugiyama 1995], angular forces (between adjacent edges at nodes \( v \)), and forces from the boundary (repulsive, bounce back).

An example of a force-directed method is the spring embedder algorithms [Eades 84; Fruchterman-Reingold 91], which consist of the following steps (see Figure 2.4):

- Replace edges with springs (zero rest length), i.e. attractive forces.
CHAPTER 2. LITERATURE REVIEW

- Replace nodes with electrically charged particles, repelling each other, i.e. repulsive forces.
- Start with a random placement of the nodes, and then just let the system go.

A spring-embedder layout method emulates this process algorithmically:

Select optimal edge length (node distance) $k$

repeat
  for each node $v$ do
    for each pair of nodes $(u, v)$
      Compute repulsive force $f_r(u, v) = -c \cdot \frac{k^2}{d(u, v)}$
    for each edge $e = (u, v)$
      Compute attractive force $f_a(u, v) = c \cdot \frac{d(u, v)^2}{k}$
      Sum all force vectors $F(v) = \sum f_r(u, v) + \sum f_a(u, v)$
      Move node $v$ according to $F(v)$
  until DONE
The force model can also be viewed as an energy system [Kamada and Kawai 1989]. If the nodes in such a system are allowed to move without restrictions, they will move to a configuration where the potential energy in the springs is minimized. In other words, an energy model is associated with the graph layout, with low energy states corresponding to a nice layout, and graph drawing is thus equivalent to the minimization of the energy. The drawing algorithm is thus an iterative optimization process. However, convergence to a global minimum is not guaranteed in all cases.
CHAPTER 2. LITERATURE REVIEW

The energy model of force-directed methods is as follows:

repeat

Compute the global energy (sum of all forces)
For all nodes (in some order) do

Check movement of the node by $d$
If improvement or random, then execute movement
Decrease the temperature

until DONE

The force-directed technique is a popular generic approach in graph layout. Variants of this approach differ in the use of different forces, definitions of the energy, cooling schedules for termination, restricting oscillations, vibrations or rotations, and methods for the minimum optimization of the energy. Some known algorithms are those of [Eades 1984; Fruchterman and Reingold 1991; Kamada and Kawai 1989]. The major advantages of force-directed methods lie in their relatively simple implementation with an intuitive concept and their flexibility, adaptability, and extensibility. In practice, force-directed methods produce good quality drawings in many graphs. Those approaches however, have some problems in aspects such as running time, termination, too many algorithm parameters. One severe problem is the difficulty in minimizing the energy function when dealing with large graphs. The convergence to the minimum, if possible at all, is very slow, particularly for very large graphs. Therefore, in general force-directed methods are limited to layout graphs with about 100 nodes.

Sugiyama Algorithm for Hierarchical Structure

Hierarchical structures can be found in a variety of applications. An example is object-oriented databases, which can be modelled as directed acyclic graphs, i.e. graphs with directed edges but no directed cycles. These graph are usually drawn in an upward fashion such that the edges “flow” in the same direction, e.g., from bottom to top. Upward drawings are more readable because the placement of nodes reflects the hierarchy. In other words, the aesthetics and conventions of hierarchical structures should be as follows: edges point downwards, avoiding long edges (i.e. few dummy nodes), few edge crossings, and many straight (vertical) edges.
One well-known algorithms for hierarchical structures is the Sugiyama algorithm [Sugiyama et al. 1981] described below.

The Sugiyama algorithm includes four phases:

1. Remove cycles
   The problem of removing cycles is NP hard [Karp 1972].

2. Compute layers
   The layer span of node \( v \) is the interval of layers, distinguished by the y-coordinates, on which node \( v \) can be placed. Dummy nodes are inserted into intermediate layers for long-span edges.

3. Reduce crossings
   The basic idea is to minimize the crossings by sweeping layer by layer, the details of which are described as:
   
   \[
   \text{repeat} \\
   \text{down phase:} \\
   \quad \text{sort next layer by barycentre or median} \\
   \quad \text{placement on lower layer} \\
   \text{up phase:} \\
   \quad \text{sort previous layer} \\
   \quad \text{placement on upper layer} \\
   \text{until DONE} 
   \]
   
   This sweeping algorithm works well and efficiently in practice although the minimization of two-layer crossings is NP-hard.

4. Assign coordinates
   All dummy nodes of a path should lie on a straight line, e.g. routing with few bends. At most, two bends for each long span edge and strict verticality between the bends integrate into the crossing minimization using heavy weights for dummy nodes and using extra space.

   There are also some other basic algorithms in the literature, such as Reingold-Tilford algorithm [Reingold and Tilford 1981] and Tutte’s Barycenter algorithm [Tutte 1963].
Besides the above basic algorithms, another direction has involved in the application of algorithms used in other areas to graph drawing. Such examples include simulated annealing [Harel 1996], self-organizing graphs [Henaux 1998; Meyer 1998], the fractal approach [Koike and Yoshihara 1993], the neural-network algorithm [Cimikowski 1996] and so on.

### 2.4 Interaction and Large Techniques

In this section, we will review several techniques for large and interactive graph layouts.

#### 2.4.1 Large Graph Layout

The force-directed methods are limited to a few hundred nodes because they have two problems: a visualization issue in which there is not enough drawing area, and an algorithmic issue in which convergence to a nice layout is too slow, as we mentioned before. For the first issue, commonly used techniques include dynamic navigation, clustering, fisheye view, and hyperbolic space. As for the algorithmic issue, several techniques for large graphs\(^3\) are recently emerging, which are based on the force-directed methods.

**Coarsening Graph**

The kind of approaches approximates a large graph by a series of *coarse graphs*. The goal of coarsening a graph is to reduce the size of the graph while keeping its crucial structure. The coarse graph is defined as:

**Definition 2.6 Coarse graph** [Harel 2000]: A sequence of graphs denoted by \(G^{k_1}, G^{k_2}, \ldots, G^{k_l}\), where \(k_1 < k_2 < \cdots < k_l = |V|\), and for all \(1 \leq i \leq l\), which are used to represent a graph \(G(V, E)\) in different abstraction levels.

\(^3\) e.g. 200 or more nodes
The combinatorial structure of a coarse graph should be significantly simplified, but important topological features of the original graph are well preserved.

The basic schema of coarse graph approaches includes coarse and fine stages. In the coarse stage, a series of coarse graphs are used to approximate the original graph. A coarser graph is then used as an initial layout for the finer graph in the fine stage (See Figure 2.4).

Figure 2.4: The process of coarse and fine graphs [from Koren et al.]

The crucial problem of these approaches is how to construct the coarse graphs from a given large graph. We will examine two main approaches below.

1. Multi Scale Graph Drawing

Hadany and Harel [Harel 2001] developed a multi-scale approach to dealing with the drawings of large graphs. This approach actually improves the force-directed method in the sense that the energy minimization is divided between coarse graphs. The
CHAPTER 2. LITERATURE REVIEW

global characteristics of a drawing are reflected in coarser graphs, so only local refinement is needed for finer graphs. The scheme of the algorithm is a recursive one consisting of:

- **Fine-scale** relocations of nodes to yield a locally organized configuration.
- **Coarse-scale** relocations to correct global disorders not found in the above step.
- **Fine-scale** relocations to correct local disorders introduced by the above step.

All actual local relocations are conducted by minimizing the energy function defined in [Kamada and Kawai 1989] using a simple gradient-descent. The coarse graph is constructed by contracting edges that minimize a convex combination of objectives. These objectives include the number of original nodes forming the new node, the degree of the nodes forming the new node in the current abstraction of the graph, as well as the number of nodes (in the coarse graph) that are adjacent to both the nodes forming the edge. This algorithm can produce good resulting layouts for large graphs with hundreds of nodes in reasonable time.

2. Multilevel graph drawing

Walshaw [Walshaw 2000] initiated a multilevel algorithm combined with an extension of Fruchterman and Reingold’s spring embedder algorithm [Fruchterman and Reingold 1991]. This approach is based on multilevel partitioning [Leland 1993], using a variant of the edge contraction heuristic. The procedure for the approach is as follows:

- Collapse a graph into a series of coarse graphs using node matching.
- Initialize the random layout of the coarsest graph $G^l$ with two nodes and one edge.
- Extend the layout from $G^{l+1}$ to $G^l$ by placing the matched pair of nodes in parents in the same position as that of its child cluster.
- Refine the layout on each graph by the standard force-directed algorithm.

At each level $l$, the force-directed algorithm is applied to graph $G^l$ using a layout refined on $G^{l+1}$ as the initial layout.
The use of the variants of Fruchterman & Reingold [Fruchterman and Reingold 1991] does not require the computation (and storage) of the all-pairs shortest path lengths. As a result, this method can handle huge graphs with up to 100,000 nodes in 10–20 minutes [Walshaw 2000].

Other approaches have also been developed to tackle large graph layouts, apart from the above approaches to coarsening graphs.

**ACE (Algebraic Multigrid Computation of Eigenvectors)**

ACE [Koren 2002] represents the energy function in a particular form and then conducts an optimal drawing of a graph by minimizing such a function. The minimum problem involved is expressed as a general eigenvalue of one that can then be quickly solved using a novel special algebraic multigrid technique. In fact, ACE uses a common technique for algebraic multigrid algorithms. These algorithms progressively coarsen a high-dimensional problem into a lower and lower dimensional one until the problem is solved exactly. A refinement process then starts with the solution being progressively projected back into higher and higher dimensions, and being appropriately updated at each scale, until the original problem is completely reproduced.

**Multi-Dimensioned Method**

The multi-dimensional method presented in [Koren 2001] includes two phases: embed a graph in a very high dimension (e.g. 50-D), and project it onto the 2-D plane using Principal Components Analysis, a well-known mathematical approach. The high-dimensional embedding is done very rapidly using a simple algorithm that utilizes the freedom given by many dimensions. This method possesses a useful inherent capability to exhibit the graph in various dimensions. This approach excels in that the running time is linear to the graph size. In practice, it is comparable to ACE.
CHAPTER 2. LITERATURE REVIEW

FADE

FADE (Force-directed Algorithm by Decomposed Estimation) [Quigley and Eades 2000] is another method for drawing large graphs, which is based on an improvement of the spring embedder algorithm.

With an initial layout method, FADE performs a geometric clustering, typically by recursive space decomposition, and then renders the drawing based on the reduced number of newly computed forces.

Roughly, FADE proceeds as follows:

- Compute initial layout
- Repeat
  - Compute edge forces
  - Construct geometric clustering
  - For each node \( u \)
    - Compute approximate nonedge forces on node \( u \) (walk the decomposition tree, testing if each internal node is "far enough" away to be considered an approximation for the graph nodes it represents)
  - Endfor
- Move nodes
- Update bounding area/volume
- Until Stopping condition is satisfied
- End Hierarchical compound graph.

In order to reduce the computational load, the nodes in a graph are clustered in a quad-tree structure, and a single force is employed to approximate the repulsive forces between a node and a cluster of distant nodes. This significantly lessens the number of forces in the graph so that the performance of FADE is improved.
CHAPTER 2. LITERATURE REVIEW

Logical Frame

*Logical frames*, proposed by Huang et al. [Huang and Eades 1998], refer to the subgraphs of a large graph, which are constructed and then displayed as needed. In other words, only a small portion of the whole large graph is shown by means of the exploration, while other parts remain invisible. Such an incremental exploration approach seems to place a visible “window” on the graph where the exploration means to move this window. Two implementation issues of this approach involve the strategies of generating new logical frames, and of repositioning these frames after each change. New logical frames may be generated in two ways, namely by adding new nodes and their neighbours [Huang and Eades 1998], and by recording the historical nodes of the user’s exploration [Huang et al.1998]. As far as the repositioning is concerned, the simplest solution is to use the same layout algorithm for each logical frame. For example, Huang et al. [Huang et al.1998] utilized a modified spring algorithm.

The advantage of such an incremental approach is that, at any given time, the subgraph to be shown on the screen may be limited in size; hence, the layout and interaction times may be no longer critical.

The *logical frame* approach for large graph layouts differs from the other approaches presented previously in that it has no limitation in the graph size, because it integrates the displaying area into the algorithmic issues as we mentioned in this section. This means that *logical frames* can be applied to visualize huge graphs that are partially unknown.

Clustered Graphs

Clustering large graphs is an alternative way to reduce visual complexity. It requires addressing underlying problems such as how to obtain a clustered graph, and how to display the clustered graph.
CHAPTER 2. LITERATURE REVIEW

For the first issue, that by which criterion (or criteria) to cluster a graph is the further subproblem. In general, there are three schemes, namely by the underlying meaning, by connectivity separators and cut methods of partition algorithms [Mattheyses 1982], or by node degrees [Batagelj et al. 1999]. As for the layout of clustered graphs, one approach proposed by Feng and Eades [Eades and Feng 1997], is the three dimensional layout. We will review several algorithms for clustering graphs in Chapters 5 and 6.

2.4.2 Dynamic Graph Layout

Dynamic graph layout is concerned with a repeated redrawing of a graph with respect to changes of the graph structure and/or some layout properties. The goal of dynamic layout is to combine both stability and readability. To this end, keeping the user’s mental map of a layout is the basic requirement. This requires minimizing the changes of the layouts so that the mental map can be preserved. It follows that how we measure the changes or how we maintain dynamic stability. A number of models for keeping the mental map have so far been presented in the literature. They can be classified into two groups, either a distance metric measuring the layout change that allows a trade-off between aesthetics with changes [Eades 1991; Tamassia 2000; Lyons 1998; North 1995], or allowed changes restricted to a subset of the nodes [Fobmeier 1997; Tollis 1998; Miriyala 1993; Paulisch 1990].

With the mental map models, a dynamic graph layout algorithm is one that, in addition to producing an aesthetic layout, has to minimize changes to the original layout. For this reason, a variety of methods have been proposed in an attempt to resolve this problem. These approaches basically consist of either restricting the changes allowed to the layout, or defining a cost function reflecting the severity of the changes and then trying to reach a good trade-off between aesthetic criteria and dynamic stability. This problem, however, still seems to be a new challenge research area.
2.3 Summary

In this chapter, we have briefly reviewed graph layout, beginning with some definitions. A graph layout is characterized by three dimension aspects in the proposed classification model. With such a classification model, we have examined several fundamental methods for different kinds of graphs, particularly the algorithms for drawing large graphs. Although those algorithms are mostly successful in practice, they still have some limitations:

- Most algorithms are developed for dealing with particular types of graphs.

In general, these algorithms serve as only a single function in most graph layout systems. In practice, such systems, however, require algorithms and approaches to resolve a set of related problems in a systematic way. Several algorithms and approaches for a proposed framework of graph layout will be particularly developed in our research, although they are also applicable to other frameworks.

- There is so far no algorithm for filtering a given graph for graph layout.

For large graphs, particularly those that are automatically extracted from an information source, one potential issue is how to filter some “noise” nodes, or how to rank nodes in terms of their roles in representation of the relations that a graph reveals. We will present an alternative method to coarsen a graph by ranking the importance of nodes in a graph.

- There are not very many algorithms available on clustering graphs for graph layout. We will particularly develop two algorithms for this purpose.

- There is a wealth of static layout algorithms, as discussed in this chapter. However, it will still be worthwhile to develop more algorithms for the dynamic graph layout. We will present our new algorithm for the dynamic adjustment of graphs in Chapter 8.
CHAPTER 2. LITERATURE REVIEW

In the next chapter, we will describe a framework for graph layout from a practical point of view. With this framework, this research will systematically develop alternative ways to deal with filtering, clustering graphs, and dynamic graphs in the following chapters.
Chapter 3

A Framework of Graph Visualization

Starting by reviewing two typical visualization models as well as a visualization architecture, this chapter proposes a new framework for graph visualization, with an emphasis on how to visually represent information in a form of graphs. This framework decomposes a pipeline of graph visualization into several stages. Those stages are vital for the remaining chapters of this thesis since they underpin the following work. The initial stage, one of the most important stages in the framework, is a graphical representation of relational data. A generic approach is proposed to construct such a representation from given premises. We demonstrate our framework by means of the implementation of a prototype called PGD.

3.1 Introduction

Reference models have been developed in various areas of information systems. These models aim to define a unique basis for system development, usage, and comparison. Many models have already been presented in information visualization. These models focus on different aspects of visualization such as visualization process, design, and guideline. In the field of graph visualization, however, there is little literature regarding such frameworks, or models of how to apply existing algorithms to real applications, although there are references to huge numbers of algorithms [Battista et al. 1994; Battista et al. 1999] and experimental results [Gansner 1997; Vismara 2000; Marshall 2001; Purchase et al. 1997]. In this chapter, we will propose a framework for practical graph visualization with the following purposes:

- To establish a conceptual framework within which applied graph visualization systems can be classified and compared.
- To describe a pipeline of applied graph visualization systems.
- To advance the description, comprehensiveness and exchange of ideas in graph visualization.
In the rest of this chapter, we review some typical existing frameworks or models on visualization, and then present a new framework based on the analytical results of characteristics of graphical applications. Following this, a generic approach for graph modelling as the first stage of such a framework is discussed in detail. An implementation of the framework called PGD is described. Finally, this chapter ends with the summary.

3.2 Related Work

3.2.1 A Task by Data Type Taxonomy for Information Visualization

Shneiderman [Shneiderman 1996] pointed out that the basic principle of visual design can be summarized as the Visual Information Seeking Mantra:

*Overview first, zoom and filter, then details-on-demand*

Using this mantra as a starting point, Shneiderman then proposed a taxonomy for the design of information visualization systems based on visualization data types and tasks. Seven types of data are identified as: 1-, 2-, 3-dimensional, temporal, multi-dimensional, tree, and network. The tasks that users can perform on such data are: overview, zoom, filter, details-on-demand, relate, history, and extract. More specifically, the seven tasks are:

- **Overview**: Gain an overview of the entire collection of items.
- **Zoom**: Zoom in on items of interest.
- **Filter**: Filter out uninteresting items.
- **Details-on-demand**: Select an item or group and get details when needed.
- **Relate**: View relationships among items.
- **History**: Keep a history of actions to support undo, replay, and progressive refinement.
- **Extract**: Allow extraction of sub-collections and of the query parameters.

Based on the user’s perspective, this taxonomy classifies seven tasks associated with data types. The users face different main tasks on various types of data. This taxonomy can be applied to sort out existing prototypes of information visualization systems and guide researchers to new opportunities.
CHAPTER 3. A FRAMEWORK FOR GRAPH VISUALIZATION

3.2.2 Information Visualization Data State Reference Model

Figure 3.1 shows an overview of the Information Visualization Data State Reference Model [Riedl 1998]. This model breaks a visualization data pipeline into four distinct stages, namely Value, Analytical Abstraction, Visualization Abstraction, and View. The transformation of these data stages requires one of the data transformation operators including Data Transformation, Visualization Transformation, or Visual Mapping Transformation.

![Image not available - see printed version](image)

Figure 3.1: Information visualization data state reference model [From Riedl 1998]

3.2.3 An Architecture for Visualization of Large Graphs

In the area of graph visualization, P. Eades [Eades 2000] proposed a layered architecture for presenting clustered graphs. This architecture contains Graph, Clustering, Abridgement, and Picture layers, as illustrated in Figure 3.2. The users can directly manipulate data in these layers by a set of operations provided. After gathering the graph data, a group of related nodes in the graph is clustered into a superimposed hierarchical clustering node. The architecture supports abridgments of clustered graphs that are logical views of parts of the clustered graph. The users can
focus on special areas of the graph by changing the abridgment. These changes are immediately reflected in the *picture* of the abridgement.

![Image not available - see printed version](image-url)

**Figure 3.2: Architecture for visualization of large graph [From Eades 2000]**

This architecture provides a suitable framework for visualizing huge graphs by a combination of clustering and navigation methods.

All those models or frameworks are user-oriented, or data (or graph) oriented. Clearly, these models provide us a picture of how data (or graphs) will be transformed and with what kinds of tasks users are confronted at each stage. However, they do not systematically discuss graph applications in real settings. Thus, from a practical perspective, what are the steps for applying the numerous existing layout algorithms to real graph applications? What are the requirements of such applications with respect to graph modelling?

In the following section, we attempt to answer those questions by proposing a new framework by which our research work is underpinned.

### 3.3 A New Framework for Graph Visualization

We start with an analysis of the characteristics of graph visualization applications and then describe the framework.

#### 3.3.1 Characteristics of Graph Visualization Applications

There are many graph visualization applications ranging from Web sites to gene visualization. The number of characteristics of graph visualization applications is thus enormous. However, our main concern here is to find the characteristics of graphs that are highly related to the techniques for graph visualization.
3.3.1.1 Attributed Graphs

As we know, a graph is usually employed to model relational objects, where the nodes correspond to objects, and the edges represent relations between the objects. A node in most traditional graph visualization algorithms is mapped into an abstract point in the Cartesian plane, occupying almost no screen space. An edge is a line connected to two nodes in a graph. The reality is, however, that every object has a set of properties and every edge is also associated with a set of attributes. In order to incorporate the semantic information about the properties of both objects and relations, *attributed graphs* or *attributed relational graphs* were proposed by Tsai and Fu [Fu 1979]. Attributed graphs, since then, have been widely used in the literature of pattern recognition. In the reviews by Bunke and Messmer [Bunke 1993; Messmer 1997], various applications had been reported, such as character recognition, schematic drawing analysis, two-dimensional shape analysis, stereo matching, interpretation of three-dimensional objects, dynamic scene analysis, and medical image analysis and chip inspection. Moreover, additional applications were described in [Walischewski 1997; Shearer 1998; Lourens 1998]. The problem of how to visualize attributed graphs should thus receive much more attention in the field of graph visualization [Huang 2001].

An attributed graph has attributes attached to each node and edge, with a formal definition as follows.

**Definition 3.1 Attributed Graph**: A 4-tuple graph \( G = (V, E, \mu, \nu) \), where \( V \) is a set of nodes, \( E \subseteq V \times V \) is a set of edges, \( \mu : V \rightarrow A \) maps the nodes to node attributes, and \( \nu : E \rightarrow A_E \) maps the edges to edge attributes, where \( A_V \) and \( A_E \) are the sets of node and edge attribute names respectively.

We then can provide a definition of *attributed graph layout*, based on the definition of *Visual Vocabulary*.

**Definition 3.2 Visual Vocabulary** ([Card et al. 1999]): A set of marks and their graphical properties used to visually encode information. Let \( M \) be marks and \( R \) graphical properties. That is, \( M = \{ \text{Point, Line, Surface, Area, Volume} \} \) and \( R = \{ \text{Position, Colour, Grey Scale, Size, Shape, Orientation, Texture} \} \), where the elements are also sets. The *Visual Vocabulary* can therefore be defined as \( Q = M \times R \).
CHAPTER 3. A FRAMEWORK FOR GRAPH VISUALIZATION

Definition 3.3 Attributed Graph Layout: A graph layout containing two mappings: one mapping is from nodes to their positions in space, \( f: V \rightarrow \mathbb{R}^n \) where \( n = 2 \text{ or } 3 \), and \( f \) is a layout algorithm. The other is formed by choosing a visual structure in accordance with the visualization design guidelines, \( r: G \rightarrow Q \), where \( Q \) is the visual vocabulary defined above, i.e., \( Q = M \times R \), and \( r \) the design rules.

Proposition 3.1 The traditional graph layout is only a special case of the attributed graph layout. Specifically, the second mapping \( r \) in a traditional graph layout simplifies to: \( V \rightarrow \{ \text{Point} \} \), and \( E \rightarrow \{ \text{Line} \} \).

A labelled graph, another kind of extensively used graph, is also a special case of an Attributed Graph. It has labels associated with each node or edge. When applying traditional graph layout algorithms to a labelled graph, the problem of overlapping nodes may arise. We thus develop an algorithm to remove such overlapping nodes in Chapter 8.

3.3.1.2 Large Graphs

When the number of nodes and edges in a graph becomes large enough, users cannot perceive all nodes and edges at the same time, and information cannot be entirely displayed on a limited display space. Various techniques have already been developed to solve this problem. Leung [Leung and Apperly 1994] broadly categorized large graphical data into distortion-oriented and non distortion-oriented presentations. A simple taxonomy of techniques for such presentations is also provided including encoding, spatial transformation (geometric), data suppressing (abstraction and threshold), zooming, windowing, as well as paging and clipping. From an algorithmic point of view, we also reviewed some techniques for coping with large graphs in Chapter 2.

The above problem with the large graph layout can alternatively be solved by removing part of the information by means of filtering or abstraction. Filtering removes some less important information, whereas abstraction forms higher-level summaries by clustering or grouping parts of information. Abstraction can also be
seen as information hiding, in which information can be gradually provided in more detail as users need.

### 3.3.1.3 Dynamic Graphs

In applications, graph drawing involves repeated redrawing, resulting from frequent changes of the graphs. Graph layout is traditionally viewed as static. However, dynamic and interactive graphs have many applications. For example, an interactive system should provide flexible facilities such as insertion or deletion of the graph, open or close of sub-graphs, and so on. In addition to producing an aesthetic layout, dynamic graph drawing requires minimizing the changes to keep intact the user’s mental map of the original layout.

### 3.3.2 A Framework of Graph Visualization

The analysis of characteristics of graphical visualization applications results in proposing a framework for graph visualization, as illustrated in Figure 3.3.

Graph visualization can be viewed as adjustable mappings from relational data, to visual form, to the users. On the top in Figure 3.3 is the information to be graphically visualized while the users on the bottom can directly manipulate the visualization. The left side arrows and the arrow between Views and Users indicate the possible adjustment of these transformations by the users. In this pipeline, the original abstract Relational Data undergoes several processing stages to obtain a graphical view.

- **Relational Data**: The data to be visualized.
- **Graph Representation**: The information is represented as a set of entities and their relationships. A graph \( G = (V, E) \) is then used to model them, where \( V \) denotes the entities, and \( E \) the relationships between the entities. In some cases, the transformation from Relational Data to Graph Representation cannot be finalized at one time. The process requires an incremental procedure. For example, in order to model the structure of a website into a graph, it is necessary to incrementally extract web pages represented as nodes and their relationships as edges.
- **Filtering**: There are two meanings of filtering. One is to remove “noise” nodes and their associated edges. In an application, the transformation from relational data
into a graph is automatically extracted by a computer program. This may produce some “noise” nodes. For this reason, the raw graph requires manipulating to filter these “noise” data. On the other hand, filtering can be used to suppress unimportant nodes and their related edges, thus highlighting the relatively important nodes in a graph, by using an adjustable threshold to control the appearance of the nodes.

**Clustering:** If a graph is too large to fit on the screen, groups of related nodes are “clustered” into super-nodes. The users see a “summary” of the graph, namely the super-nodes and super-edges between the super-nodes. Some clusters may be shown in more detail than others. The process of clustering involves discovering groups in the data. In a case of graph visualization, clustering means finding a set of relatively highly connected nodes and their associated edges in a graph, and then using a special type of node called a *meta-node* or *super-node* to represent them. In other words, the clustering techniques make it possible to represent a graph by displaying fewer elements, allowing users to control the level of detail by ‘opening’ and ‘closing’ meta-
CHAPTER 3. A FRAMEWORK FOR GRAPH VISUALIZATION

nodes. The clustering approach has been taken by a number of graph drawing researchers [Crouch et al. 1989; Dongen 2000; Eades and Feng 1996; Everitt 1974; Fowkes and Mallows 1983; Huang and Eades 1998; Kaufmann and Wagner 2001].

**Layout:** This is applying an algorithm to layout a given graph.

**View:** The next problem addressed is how to present the information in different ways. The users need to interact frequently with the graph, changing the views in order to gain insight into the data. All these features require a system that can easily adapt to the users’ needs and quickly change the way that a graph is presented. These enable the users to freely explore a graph, in addition to navigating it. The users should also be allowed to interactively apply criteria that result in different sets of clusters within the data.

The transformation mappings in Figure 3.3 can be formalized as follows:

\[ \{\text{Entities, Relationships}\} \xrightarrow{f_1} G(V, E) \xrightarrow{f_2} G'(V', E') \xrightarrow{f_3} G^C(V^C, E^C) \xrightarrow{f_4} f_5 \xrightarrow{f_6} \{l_1, l_2, \cdots, l_n\} \]

where

- \( f_1 : D \rightarrow G \) called *Data Transformation*, which illustrates a relational information data in a form of graph \( G \).
- \( f_2 : G \rightarrow G' \) where \( |G'| \leq |G| \) called *Filtering Mapping*, which reduces the size of graph \( G \) by removing some unimportant nodes and associated edges.
- \( f_3 : G' \rightarrow G^C \) called *Clustering*, which groups relatively high connected subgraphs and forms a course graph with the reduced size.
- \( f_4 : G^C \rightarrow L \) called *Layout*, i.e. a layout algorithm for creating a layout \( L \) that is a mapping from \( G^C \) into a 2D or 3D plane.
- \( f_5 : L \rightarrow \{l_1, l_2, \cdots, l_n\} \) called *View Transformations*, where \( l_1, \ldots, l_2, l_n \) represent different views of the layout \( L \). The users see only a small subset of the nodes and edges at any one time, and facilities are provided to navigate through the graph. Common techniques include Zooming, Overview+Detail, and Focus+Context [Card et al. 1999; Hara et al. 1996; Lamping et al. 1995]. The users can control viewpoints, distort the layout, and navigate the graph.
Overall, the mapping can be represented as $f : D \rightarrow \{l_1, l_2, \ldots, l_n\}$ where

$$f = f_5 \circ f_4 \circ f_3 \circ f_2 \circ f_1$$

The Filtering and Clustering stages are optional within the above framework, if the size of a graph is small. Moreover, the Filtering stage can be omitted in some applications. In other words, the Filtering and Clustering deal mainly with large graphs. In addition, the View stage is highly related to the Layout stage, since each view may result in a re-layout of a graph.

The goals of our work in the following section and chapters 4, 5, 6, 7 and 8 are to principally find appropriate mapping functions $f_2, f_3, f_4$, and $f_5$, respectively.

Motivated by applying existing algorithms to applications, we proposed the framework based on the general requirements of real graphical applications. It is this feature that mainly distinguishes it from other models described previously.

Graph modeling, as the first and important stage of the framework, will be addressed in the following section.

### 3.4 Graph Modelling

In the above framework, an important issue involves how to model relational data into a graph. This problem can further be divided into two sub-problems, namely how to model an object into a node, and how to model the relationship between two objects into an edge. The former just simply maps an object into a node, while the latter is not so easy. In a non-attributed graph, an edge is directly built between two objects if they have one common attribute. Two Web pages, for instance, can be represented as two corresponding nodes, and an edge is created in the Web graph if these two pages have a hyperlink between them. The common attribute used to build an edge in this example is the hyperlink. In real scenarios, the relationships between two objects (nodes), however, may be multiple. Two Web pages may be hyperlinked, share several common keywords, and refer to the same references, etc. In an attributed graph as shown in Definition 3.1, the edges are associated with a set of attributes, in which every member represents one aspect of the relationships between two corresponding nodes. In other words, each attribute of an edge is determined by a
relationship between one or a few attributions of two corresponding nodes connected with such an edge. Equivalently, an attribute of an edge is determined by a combination of one or more attributes of corresponding nodes, rather than only an attribute such as the hyperlink shown in the above example. Thus, the problem addressed here is how to find the mappings from edge attributes to the two involved node attributes. We will give a general approach to determining the mappings.

### 3.4.1 Node Attribute

According to Definition 3.1, the nodes in an attributed graph have a set of domain-specific attributes in which each member describes one characteristic of nodes. In order to represent the relations between nodes, the corresponding attribute values should be given. In other words, an attributed graph is supposed to be associated with two sets: an attribute name set, such as \( NA_1 = \{\text{Hyperlink, Keyword1, Keyword2}\} \), and an attribute value set, such as \( A_1 = \{1, 1, 2\} \), where the values correspond to the attribute names. The attribute value set can be treated as a local feature vector, namely \( A_1 = (1, 1, 2) \). Our purpose is to build the relation between two nodes based on their vectors. However, different nodes may have various attribute names and corresponding values. For example, two Web pages, represented as two nodes, have their attribute name sets \( NA_1 = \{\text{Hyperlink, Keyword1, Keyword2}\} \) and \( NA_2 = \{\text{Reference, Keyword1}\} \), as well as their attribute value sets \( A_1 = \{1, 1, 2\} \) and \( A_2 = \{1, 2\} \). In this case, the first page has one Hyperlink, one Keyword1, and two Keyword2s, while the second page has no Hyperlink, one Reference page, and two Keyword2s. Obviously, these two node local feature vector \( A_1 = (1, 1, 2) \) and \( A_1 = (1, 2) \) have different dimensions and the corresponding various attribute names. This means that we cannot directly build the relations between the two nodes based on their attribute values. Actually the following two observations are true for most cases. First, the number of members in each set of node attribute names may differ due to the fact that different nodes have various features. Second, different sets of node attribute names may contain common attribute names. To resolve this problem, we need a unified attribute name set called the global attribute name set to include all the possible attribute names within a graph. It follows that each node global feature vector with the same dimension is easily obtained by comparing the elements of its node attribute
name set with those of the global attribute name set. Suppose that given the heterogeneous description of node attribute name sets $A_1, A_2, \ldots, A_{|V|}$, we have a global attribute name set:

$$A = \bigcup_{i=1}^{|V|} A_i$$

Clearly, $|A| \geq |A_i|$ holds, where $1 \leq i \leq |V|$.

Note that the attribute names fall into two categories, namely nominal one measured by a binary number (1 or 0), and cardinal one by a real number. For example, here we consider the Hyperlink as a nominal attribute, which tells whether a node has the Hyperlink attribute or not. The corresponding value is 1, if it has or 0 otherwise. In contrast, the Keyword attribute is cardinal, meaning that it can has two or more Keywords. The node attribute name set $A_i$ can thus be written as $A_i = N_i \cup O_i$.

That is, a node attribute name set $A_i$ consists of a nominal attribute name set $N_i$ and a cardinal attribute name set $O_i$.

Here is an example. Suppose that the node attribute name sets of two Web pages are as follows:

$$A_1 = \{\text{Hyperlink}, \text{Reference}, \text{Keyword1}, \text{Keyword2}, \text{Keyword5}\}$$
$$A_2 = \{\text{Hyperlink}, \text{Keyword1}, \text{Keyword3}, \text{Keyword4}, \text{Keyword5}\}$$

Obviously, we have:

$$N_1 = \{\text{Hyperlink}, \text{Reference}\}$$
$$O_1 = \{\text{Keyword1}, \text{Keyword2}, \text{Keyword5}\}$$
$$N_2 = \{\text{Hyperlink}\}$$
$$O_2 = \{\text{Keyword1}, \text{Keyword3}, \text{Keyword4}, \text{Keyword5}\}.$$  

Also, we have the node global attribute name set $A$ consisting of:

$$A = A_1 \cup A_2 = \{\text{Hyperlink}, \text{Reference}, \text{Keyword1}, \text{keyword2}, \text{keyword3}, \text{Keyword4}, \text{Keyword5}\} = N \cup O = \{\text{Hyperlink}, \text{Reference}\} \cup \{\text{Keyword1}, \text{keyword2}, \text{keyword3}, \text{Keyword4}, \text{Keyword5}\},$$

where the first two members in $A$ ($|N| = 2$) are nominal, and the rest are cardinal.

In the following, we give an approach to converting a node local feature vector into a node global feature vector so that all node feature vectors have the same dimension.

Define an $|A|$-dimensional vector

$$A = (a_1, a_2, \ldots, a_{|A|})$$
as the corresponding vector of the node global attribute name set $A$. Similarly, every node has a node global feature vector quantitatively describing its characteristics. Using the set $A$ as a benchmark, an $|A|$-dimensional, global attribute feature vector $A_i$ of node $v_i$ is derived from its node attribute name set $A_i$.

Let $a_i(k) \ (1 \leq k \leq |A|, 1 \leq i \leq |V|)$ be the $k$-th component of a node global feature vector $A_i$ of node $i$, $w(k)$ a component of its local feature vector that corresponds to the $k$-th attribute name in node global attribute name set $A$, $NA_i$ its node attribute name set, and $A(k)$ the $k$-th member of $A$. We then have:

$$a_i(k) = \begin{cases} w_i(k) \ |^p & \text{if } A(k) \in NA_i \\ 0 & \text{if } A(k) \notin NA_i \end{cases} \quad (3.1)$$

where

$$p = \begin{cases} 0 & \text{if } 1 \leq k \leq |N_i| \\ 1 & \text{if } |N_i|+1 \leq k \leq |A| \end{cases} \quad \text{and } 1 \leq i \leq |V|$$

The first $|N|$ members of the set $A_i$ are nominal, and the rest are cardinal.

The correctness of the above equation is based on the assumption that a node attribute name set $A_i$ consists of the number of $|N|$ nominal attribute names and of $|O|$ cardinal ones in sequence. By comparing each element of the node global attribute set $A$ with that of the node attribute name set $NA_i$, the component $a_i(k)$ of the Node Global feature Vector $A_i$ is 1 if the $A(k)th$ nominal attribute name is present in $A_i$, or 0 otherwise. For each rest component $a_i(k)$ directly obtains the corresponding value $w(k)$ in its local feature vector if this cardinal attribute name is present in $A$, or 0 otherwise.

We can rewrite it as a definition.

**Definition 3.4 Node Global Feature Vector:** A vector of quantitative description of the characteristics of a node. A node global feature vector $v_i$: $A_i = (a_i(1), a_i(2), \ldots, a_i(|A|))$, derived from its node attribute name set $NA_i$, and the node global attribute name set $A$, where the components $a_i(k) \ (1 \leq k \leq |A|, 1 \leq i \leq |V|)$ are calculated by the formula (3.1).

The Node Global Feature Vectors of two Web pages in the previous example are obtained by comparing $A$ with $A_1$, and with $A_2$:

$$A_1=(1,1,2,3,0,0,5), \ A_2=(1,0,1,0,6,5,8)$$

Although different nodes have various node attribute name sets, they are now associated with node global feature vectors with the equal dimension that is determined by the number of members in the node global attribute name set. This
provides a uniform basis for building the relationships between nodes with various attributes.

**3.4.2 Building Relationships among Nodes**

So far, we have discussed the node attribute set and its feature vector. Next we will discuss edges that represent more than one relation defined on pairs of nodes.

Similarly, we use an edge global attribute set $E$ that unifies the attributes of all the edges in a graph:

$$ E = \{ e_i | 1 \leq i \leq m \} $$

where $m$ is the number of distinct attributes of edges in a graph. That is, there are $m$ possible kinds of relations between pairs of nodes. Each kind of these relations can be separately represented as a graph or directed graph, where the edges in various graphs are associated with different relations. However, examining such $m$ graphs in total can be difficult. An alternative way is to simultaneously present the $m$ graphs in a single graph where points represent nodes, and lines and their visual attributes indicates the $m$ relations. In the above example, suppose $E = \{ \text{Link, Reference, Similarity} \}$, where

- **link** refers to whether or not a Web page has a hyperlink to another,
- **reference** indicates whether or not two pages refer to the same references, and
- **similarity** is the degree of similarity between the two pages in terms of how many common keywords they have.

We can use a line, line arrows and the width of a line in one graph to characterize Link, Reference and Similarity relations between nodes, respectively. In a case where two Web pages have a hyperlink between them, but do not share common reference and keywords, we should draw a non-arrow line with a normal width between the two nodes. This means that an edge and its visual attributes such as the line type, width, and color, in the drawing of an attribute graph, are depend on its two corresponding node attributes. Using an edge and its visual attributes to encode multiple relations between two nodes, it follows that which node attributes or their combination determine what kinds of the edge attributes. We will address this question in the following.

Also, we need to construct the corresponding vector: $E = (e_1, e_2, \ldots, e_m)$. 
In line with Definition 3.1 and the above description, we can formalize the relations between the node global attribute set and the edge set as $A \times A \rightarrow E$, which means node attributes determine edge attributes. In particular, let $S_i$ be a subset of set $A$, i.e., $S_i$ is a derived set of an attribute or a combination of attributes from $A$, then $S_i \times S_i \rightarrow \{e_i | 1 \leq i \leq m\}$ where $S_i \subseteq A, 1 \leq i \leq m$. That is, every node has $m$ attribute subsets that are used to build $m$ relations with others. For example, given $A = \{\text{Hyperlink, Reference, Keyword1, Keyword2}\}$ and $E = \{\text{Link, Reference, Similarity}\}$, we can assume that $\text{Hyperlink} \times \text{Hyperlink} \rightarrow \text{Link}$, $\text{Reference} \times \text{Reference}$ (in the $A$ set) $\rightarrow \text{Reference}$ (in the $E$ set) $\text{(Keyword1, Keyword2)} \times (\text{Keyword1, Keyword2}) \rightarrow \text{Similarity}$.

We can accordingly construct the corresponding vectors $S_{i}^{v}$ and $S_{i}^{u}$ $(1 \leq i \leq m; v, u \in V)$ of two nodes $v$ and $u$. Here $S_{i}^{v}$ and $S_{i}^{u}$ are subset-vectors of node feature vectors $A_v$ and $A_u$, respectively.

With vectors $S_{i}^{v}$ and $S_{i}^{u}$, we can subsequently compute the node-node similarities by means of certain similarity measurements. The purpose of this measurement is to determine whether the $i$-th relationship between these two nodes is strong enough to be visually represented in an edge connecting them.

The elements of vectors $S_{i}^{v}$ and $S_{i}^{u}$ can be viewed as the coordinates in an $n$-dimensional space ($n = |S_{i}^{v}|$). Especially, with $n = 2$, they can be indicated as two points in a two-dimensional Cartesian plane, and then be compared to each other by employing the widely used cosine measurement. If the cosine value exceeds a threshold, this kind of relationship between two nodes can then be created in a form of an edge or an attribute of the edge. For example, suppose that the coordinates of $S_{i}^{v}$ and $S_{i}^{u}$ in two-dimensional space are $(x_v, y_v)$ and $(x_u, y_u)$, respectively. The angle difference between these two vectors is $\vartheta$ as shown in Figure 3.4. We select 0.90 as the cosine threshold, namely if $\cos(|\vartheta|) \geq 0.90$, and then this type of the relationship between two nodes will be illustrated in the graph layout.
Figure 3.4: Example of cosine threshold

Generally, the cosine similarity between two vectors $S_i^v$ and $S_i^u$, denoted by $\theta_{iw}^i$, is defined as:

$$\cos \theta_{iw}^i = \frac{S_i^v \cdot S_i^u}{||S_i^v||_2 ||S_i^u||_2}$$

Based on this, the $k$-th component of vector $E$ is computed as follows:

$$e_k = \begin{cases} 
\min \{ ||S_k^v||, ||S_k^u|| \}^p & \text{if } \theta_{iw}^i \geq \delta_k \\
0 & \text{if } \theta_{iw}^i < \delta_k 
\end{cases} \quad (3.2)$$

and

$$p = \begin{cases} 
1 & \text{if } S_k^v, S_k^u \in O \\
0 & \text{otherwise} 
\end{cases}$$

where:

1. $1 \leq k \leq m$ and $\delta_k$ is a threshold.

2. For the cardinal attributes (its set denoted as $O$), we use the minimum corresponding vector norm, i.e. $\min \{ ||S_i^v||_2, ||S_i^u||_2 \}$, as the degree of one kind of relationship between two nodes $v$ and $u$, where the norm of a $n$-dimension vector is $\sqrt{s_1^2 + s_2^2 + \cdots + s_n^2}$.

Summarily, Figure 3.5 illustrates the transformation relationships described above in the course of the modeling.

**Proposition 3.2** Suppose that the global attribute edge vector is $E = (e_1, e_2, \ldots, e_m)$ ($1 \leq m \leq |A|$) and its sub-vectors are $S_i (S_i \subseteq A, 1 \leq i \leq m)$. Given nodes $v, u \in V$, then their associated attribute edge vector is $E^{uw} = (e_1^{uw}, e_2^{uw}, \ldots, e_k^{uw}, \ldots, e_m^{uw})$ where a component $e_k^{uw}$ is computed by equation 3.2.
3.4.3 The Algorithm for Graph Modelling

The pseudo code of the algorithm for graph modelling is as follows:

**Input:** Node attribute Name sets \( A_1, A_2, \ldots, A_{|V|} \) and relationships between node attribute subsets and edge attributes, i.e., \( S_1 \times S_i \rightarrow \{ e_j \mid 1 \leq i \leq m \} \), and the thresholds \( \delta_1, \delta_2, \ldots, \delta_m \).

**Output:** An edge attribute vector \( E \)

// Find the node global attribute set \( A \)
\[ A = \phi \]
for \( i = 1 \) to \( |V| \)
\[ A = A \cup A_i \]
end for
\[ A = N \cup O \]
// construct \( m \) node attribute vectors for each node \( v \)
for each \( v \in V \)
    for \( i = 1 \) to \( m \)
        \[ S_i^v = (0, 0, \ldots, 0)_{|S_i|} \]
        for \( k = 1 \) to \( |S_i| \)
            if \( A_i \cap s^v_i (k) \neq \phi \) then
                if \( s^v_i (k) \in N \) then \( s^v_i (k) = 1 \)
In the above algorithm, $s_i^v(k)$ and $e^{vw}(k)$ represent the $k$-th component of vectors $S_i^v$ and $E^{vw}$, respectively. The complexity of this algorithm is $n^2$.

### 3.4.4 An Example

Referring to the preceding example, the node attribute set and attribute value sets of two Web pages are:


\[ N_A_1 = \{ \text{Hyperlink, Reference, Keyword1, keyword2, Keyword5} \}, A_1 = \{ 1, 1, 2, 3, 5 \}, \]
and \[ N_A_2 = \{ \text{Hyperlink, Keyword1, keyword3, Keyword4, keyword5} \}, A_2 = \{ 1, 1, 6, 5, 8 \} \], which consist of two sub-sets, respectively: \[ N_1 = \{ \text{Hyperlink, Reference} \}, O_1 = \{ \text{Keyword1, keyword2, Keyword5} \}; \text{ and } N_2 = \{ \text{Hyperlink} \}, O_2 = \{ \text{Keyword1, keyword3, Keyword4, Keyword5} \} \]. Thus, the node global attribute name set is \[ A = \{ \text{Hyperlink, Reference, Keyword1, keyword2, keyword3, Keyword4, Keyword5} \} \].

The corresponding two Node Global Feature Vectors with a dimension \(|A|\) are obtained by comparing \(A\) with \(A_1\) or \(A_2\) according to Equation (3.1):

\[ A_1 = (1, 1, 2, 3, 0, 0, 5), \text{ and } A_2 = (1, 0, 1, 0, 6, 5, 8) \]

Suppose \(E = \{ e_1, e_2, e_3 \}, \)

\[ S_1 = \{ \text{Hyperlink} \}, S_2 = \{ \text{Reference} \}, \]
\[ S_3 = \{ \text{Keyword1, Keyword2, Keyword3, Keyword4, keyword5} \} \]

then we have \(S_1 \times S_2 \rightarrow \{ e_i | i = 1, 2, 3 \} \). The first two components of vector \(E\) are \(e^{A_{A_1}}(1) = 1\) and \(e^{A_{A_2}}(2) = 0\). Two cardinal sub-vectors of \(A\) are \(S_3^{A_1} = (2, 3, 0, 0, 5)\) and \(S_3^{A_2} = (1, 0, 6, 5, 8)\) According to Equation (3.2),

\[ \theta_{i_i} = (2 \times 1 + 3 \times 0 + 0 \times 5 + 5 \times 8) / \left( \sqrt{2^2 + 3^2 + 0^2 + 5^2} \times \sqrt{1^2 + 0^2 + 6^2 + 5^2 + 8^2} \right) \]
\[ = 0.607 > \delta_i \text{ holds, if } \delta_i = 0.50. \text{ So we have } E^{A_{A_1}} = (1, 0, 0.607). \]

Suppose that the Visual Vocabulary for this particular example is \(Q = \{ \text{line, yellow, width} \}\). Due to the presence of the first and third components of vector \(E^{A_{A_2}}\), we should draw a line with its width proportional to 0.607 as the edge between two nodes indicating the two web pages \(A_1\) and \(A_2\), where the number 0.607 indicate the degree to which they are similar, with respect to having some common keywords. The color of this line, however, should not be yellow since the two web pages have no common reference.

### 3.5 PGD: A Prototype of the Framework

In order to demonstrate the effectiveness of our algorithms and approaches, we developed a prototype for the proposed framework called PGD (a framework for practical graph drawing), employing the objected-oriented design.
The user interface of PGD is shown in Figure 3.6. The screen is divided into three parts.

On the top are several menu options: File and Edit for loading and saving the layout of a graph and for deleting selected nodes and edges; Filtering for removing the “noise” or unimportant nodes and edges in a graph; Clustering for abstracting the graph into multiple coarse sub-graphs; Layout for assigning the locations to both nodes and edges; Adjustment for the dynamic layout of a graph, and View for specifying the parameters and interacting with the graph and its layout.

In the middle is a toolbar, which is a collection of buttons, one for each node type and one for each edge type. Exactly one of the buttons is active at any time.

On the bottom is the graph drawing area. Users can use the mouse to create, select, and move nodes and edges.

Overall, the current prototype can obtain a graph by either of three methods described later. The prototype then processes the obtained graph by filtering and clustering. Following this, an algorithm is chosen to layout the graph. Finally, the layout of the graph can be interactively changed by the users.
CHAPTER 3. A FRAMEWORK FOR GRAPH VISUALIZATION

The architecture of PGD consists basically of several modules: Input module, Filter module, Cluster model, Layout Module, and View module. Each module is described as follows:

**Input Module**: This is used to extract the entities from a relational data set, and then analyze the relationships between them, and finally construct a graph representation \( G = (V, E) \). In PGD, three ways were implemented to obtain the graph:

- Create a graph from scratch using the tools provided in the prototype.
- Input directly from an external file containing the specifications of a graph by using the “import” submenu at the “File” menu.
- Extract the Web pages from a Web site by using a program called WebCrawler, the interface of which is as follows:

![Figure 3.7: Another interface of the PGD](image-url)
CHAPTER 3. A FRAMEWORK FOR GRAPH VISUALIZATION

At the beginning, the WebCrawler requests the user to input the valid http address of a Web site and the depth of exploration, that is, a number representing the search limit. It then extracts and examines all the linked URLs, and processes the search recursively until reaching the specified limit, or stop automatically if it encounters a dead end.

The pseudo-code summary of the program is simple:

- Load the page of the starting URL
- Validate the page and make sure it is not already searched
- Extract all URLs contained in the current page
- Repeat the above procedure for all found URLs.

The WebCrawler outputs a text file in which each line contains the URLs of two hyperlinked web pages. This URL text file can be either imported into our prototype to construct a web graph and then display it, or converted into a graph layout within the WebCrawler.

**Filter module:** This module has two functions: remove “noise” nodes and their associated edges, or produce different overviews of a graph by changing a threshold. Those nodes that play less important roles in the graph are suppressed. We will discuss the filtering algorithms in Chapter 4.

**Cluster module:** Implements two clustering algorithms: KNN and SMK, to cluster large graphs, described in Chapters 5 and 6.

**Layout Module:** Applies one of the layout algorithms to present a processed graph. Currently we use the spring algorithm.

**View Module:** Users can choose different view models of a layout graph and interact with those views. It is possible to have several different views of one or more graphs, with each in its own window. There are three view modules in our prototype, namely Normal View model, Fisheye View model, and Multiple Windows Multiple Level view. The Normal View model displays a graph in a normal way with an applied
algorithm. *Fisheye View* model utilizes a fisheye algorithm to filter and represent a graph. The *Multiple Windows Multiple Level view* presents a graph in different windows with various abstract levels. The users can manipulate and navigate the graphs, adding or removing a node or an edge to arrive at their own “views” of the information.

It is easy to add a new layout algorithm and reasonably simple to add other new functions such as a parser, or even an entire new *View* module into the prototype.

These modules described above match roughly to the graph visualization framework proposed in Figure 3.3. The modules can be assembled into a graph visualization application in which each module corresponds to each processing stage of graph data. It is also possible, of course, to make use of the modules independently such as *Filtering* module, *Clustering* module, and *Layout* module.

### 3.6 Summary

In this chapter, we have presented a framework for graph visualization. This framework is motivated by investigation of the characteristics of graph visualization applications. As the first stage of the framework, graph modelling is imperative so that a generic approach is presented, with an illustrative example. The following chapters will further examine the issues involved in the presented framework.
Chapter 4

NodeRank: Filtering for Layout

When the amount of information in visual range exceeds a certain size, users cannot perceive all elements at the same time. This problem can be solved by removing parts of the information through the processes of Filtering and Clustering. In this chapter and the following two chapters, we will present these two approaches in the context of graph visualization. Both approaches have the identical purpose of reducing the number of nodes and edges within a layout. Filtering, however, achieves this by suppressing part of a graph, while clustering utilizes imposed nodes and edges to replace groups of nodes and edges.

This chapter deals mainly with the issues in Filtering as included in the proposed framework in Figure 3.3. Part of the content of this chapter has been published in [Huang and Lai 2003].

4.1 Introduction

Although there are other ways to tackle the layout of large graphs - such as clustering, navigation, and zooming – Filtering, the proposed approach, is significantly different to these. Navigation and zooming cannot replace filtering, since filtering reduces the size of a graph. Filtering serves as one alternative way of efficiently reducing visual complexity.

The reasons why a graph should be filtered are as follows:

- A graph contains “noise” nodes in the sense that those nodes interfere with the representation of a relation. Typically, a large graph is automatically generated from an information source. This may unavoidably lead to the creation of “noise” information. For example, the use of a WebCrawler program to construct a Web graph easily extracts some unwanted image files, together with html Web pages, from a Web site.
CHAPTER 4. NODERANK: FILTERING FOR LAYOUT

- Overviews first, zoom and filter, then details-on-demand. According to the Visual Information Seeking Mantra [Shneiderman 1996], as indicated in the framework shown in Figure 3.1 of Chapter 3, filtering is an efficient method of handling information on a large scale. It initially presents an overview and then gradually reveals details of the data.

Huang et al. [Huang et al. 1998] initiated to apply several rules in order to filter a real Web graph into a simplified tree for a simplified layout. These rules include: graph structure-based rule, Web context structure-based rule, information-based rule, document structure-based rule, and link number-based rule. The use of these rules ensures that the converted graph is a tree, and simplifies the original graph (see Figure 4.1). These rules are obviously concerned about both the structure and content of a Web graph.

Simple filtering techniques based on node attributes are also reported in Henry’s thesis [Henry 1992].

The above approaches have some limitations. The approach of Huang et al., for instance, is confined to a specific type of graph, i.e. the Web graph, and does not suffice to handle other types of graphs. It is necessary to develop a more sophisticated technique for filtering general graphs.

The approaches to filtering graphs can be roughly classified into structure- and content-oriented. The structure-based approach utilizes the linkages of a graph, while the content-based one uses the semantic contents of what the nodes represent. The proposed approach is a structure-based one and thus has the advantage of being applicable to any type of graphs.

Our contributions in this chapter lie in proposing a new method of filtering a graph based upon existing approaches: describing two filtering modes, and conducting experiments.
4.2 NodeRank: Node Importance Rank

Graph filtering refers to the selection of particular nodes and edges from a graph according to whether their attribute values are within a specified range. This selection may be done manually or automatically in accordance with a model of degree of interest or criteria. The key issue in relation to filtering is determining which attribute(s) qualify for distinguishing nodes and edges in a graph. A node has many attributes, such as the importance of its “role” in the graph, the content of what the node represents, the size of the node, and so on. As mentioned in Chapter 3, nodes are employed to represent objects, and edges represent relationships between these objects. In other words, the purpose of a graphical representation is to visually reveal complex relationships between objects in order to reduce the user’s cognitive load. Various nodes and edges, however, play different roles in a graph with respect to revealing such relationships. Some are prominent whereas others are trivial. Prominent nodes are those that are extensively involved in relationships with other nodes. This involvement makes them more visible to the others. The node “dept.html” in Figure 4.1, for example, is a prominent one, while the node “w9.gif” is not. For this reason, it is necessary to develop a novel method to accurately identify these prominent nodes in a graph. A function is needed to measure the
importance roles of nodes with respect to the depiction of node relationships. Fundamental to our approach is the notion of NodeRank, which is defined as follows:

**Definition 4.1 Node Importance Score**  
*(NodeRank)*: A real number indicating the degree of the importance role which a node plays in a graph, or of the involvement of a node in other node relationships. Let $G = (V, E)$ be an undirected and connected graph, and $R$ be a function that assigns a real value ranging from 0 to 1 to every node $u$ in $G$. That is, $R: V \rightarrow [0, 1]$. The *Node Importance Score* $R(u)$ of node $u$ $(0 \leq R(u) \leq 1)$, is used to rank nodes.

With this definition, it follows that we need to develop a method for the calculation of a *NodeRank*. Fortunately, the measure of “centrality” used in social network analysis [Faust 1995] provides the basis for this calculation. The concept of Centrality seeks to quantify the prominence of a node within a network by summarizing structural relationships among the nodes.

We will develop a method to measure the importance of nodes in terms of their roles within a graph. Two approaches to calculating the *NodeRank* score are discussed in the following sections. Section 4.4 gives two models of filtering graphs, followed by presenting an algorithm in 4.5. Several examples are described in Section 4.6, and conclusion is given in Section 4.7.

**4.2.1 Degree, Closeness and Between Centrality Index**

As mentioned in Definition 4.1, a function $R$ is to be constructed to map every node in a graph into a real value which can indicate the importance role of a corresponding node within the graph.

Centrality in a social network refers to the importance of a particular node in a network. Measures of centrality have been developed to “attempt to describe and measure properties of ‘actor location’ in a social network” [Faust 1995]. The importance of a node in a graph can be measured by how easily it reaches other nodes and how often it is directly connected to other nodes. In other words, The *Node Importance Score* of a node
is determined by how it directly and indirectly connects to other nodes in the graph. In this view, the role of a node can be a function of its position in a given graph.

There currently exist a variety of centrality measures, which can be roughly classified into three major types: degree, closeness, and betweenness centrality [Faust 1994; Freeman 1978; Friedkin 1991; Granovetter 1973; Katz 1953; Scott 2000]. Each of them is defined as follows.

**Definition 4.2** [Freeman 1978] *Degree Centrality*: The number of edges attached to a node $u$.

Obviously, Degree centrality can be normalized to range from 0 to 1, where 0 means the smallest possible and value 1 the highest possible centrality. The normalized measure is called a relative measure of centrality:

$$C_D(u) = \frac{\text{deg}(u)}{n-1}$$

where $\text{deg}(u) = |\{v \mid (u,v) \in E \land v \in V\}|$, and $n=|V|$.

Degree centrality reflects the direct relationships of a node with others in a graph. It also indicates the number of links connecting adjacent nodes to a local node, so it is a local centrality.

**Definition 4.3** ([Freeman 1978]) *Closeness Centrality*: The sum of the shortest paths, defined as the number of edges connecting two nodes, between a node $u$ and all other nodes.

$$C_C(u) = \frac{\sum_{v \in V} d(u,v)}{n-1}$$

where $d(u,v)$ is the shortest path between nodes $u$ and $v$. Measures of centrality based on closeness reflect a node’s freedom from the controlling relationships of others and its capacity for independent relationship within the graph. This measure actually indicates how far a node is from all others. A node with a higher closeness score is less centralized than one with a lower closeness score. The most central nodes can quickly interact with all the other nodes because they are close to them.
Definition 4.4 ([Freeman 1978]) Betweenness Centrality: The ratio of the number of shortest paths between two nodes passing a node \( u \) to the number of all possible such shortest paths in a graph:

\[
C'_b(u) = \sum_{i=1}^{n} \sum_{k=1}^{i-1} \frac{g_{jk}(u)}{g_{jk}}, \quad C_b(u) = \frac{C'_b(u)}{(n-1)(n-2)/2}
\]

where \( g_{jk} \) is the number of shortest paths from node \( j \) to node \( k \), and \( g_{jk}(u) \) is the number of those shortest paths that include node \( u \).

Betweenness Centrality reflects the intermediate location of a node along indirect relationships linking other nodes. It “measures the extent to which a particular point lies between the various other points in a graph: a point of relatively low degree may play an important ‘intermediary’ role and so be very central to the network” [Scott 2000]. A node with a high betweenness has the capacity to facilitate or limit interaction between the nodes it links.

Both Closeness and Betweenness are global centralities. Note that all the above measures are relative ones, with the corresponding possible maximum values as denominators in the equations.

In an example of the graph in Figure 4.2(a), the above three centralities of node \( N_3 \) are computed as follows:

Degree: \( C_D(N_3) = 3/(7-1) = 0.5 \)

Closeness: \( C_C(N_3) = \sum_{i=1}^{7} d(N_3, N_i) / (7-1) = (1+1+2+1+2+2)/6 = 1.5 \)

Betweenness:

\[
C'_B(N_3) = \sum_{i=1}^{7} \sum_{k=1}^{i-1} \frac{g_{jk}(N_3)}{g_{jk}} = (2 + 3 + 3 + 3 + 3)/(1+2+3+4+5+6) = 14/21
\]

\[
C_B(N_3) = 2C'_B(N_3) / (7-1)(7-2) = 2/45
\]

We observe that these three centrality measures may produce contrary results for the same graph. It can be a case in which a node has a low degree centrality, but with a high betweenness centrality. Freeman [Freeman 1978] demonstrated that the betweenness centralities best capture the essence of important nodes in a graph, while degree centralities appear to produce the smallest node variances. In order to overcome the
drawbacks of single centrality, the combination of degree, closeness, and betweenness yields the following measure as the NodeRank score:

\[ R_n(u) = w_1 C_D(u) + w_2 C_c(u) + w_3 C_B(u) \]

\[ = w_1 \frac{\deg(u)}{n-1} + w_2 \frac{\sum_{v \neq u} d(u, v)}{n-1} + w_3 \sum_{j=1}^{n} \sum_{k=1}^{j-1} g_{jk}(u) \left/ \frac{g_{jk}}{(n-1)(n-2)/2} \right. \]  

(4.1)

where the weights \( w_1, w_2, \) and \( w_3 \) sum to 1. For simplicity, we can give equivalent importance to the three measures by assigning equal values of the weights in our experiments. By adjusting the values of these weights, the emphasis can be placed on different measures. In general, a node will have a high NodeRank score, if it has a high degree, is easily accessible to (close to) all other nodes, and lies on several shortest paths between other nodes.

The time complexity of these measures is obviously \( O(n^2) \), meaning that the NodeRanks for large graphs are computed offline.

### 4.2.2 Eigenvector Centrality

Degree, Closeness, and Betweenness centralities are derived from graph distances. Nodes are regarded to be important on the basis of how close they are to other nodes in a graph. However, at the same time we should consider the importance of nodes that are proximate to the node under study. Being chosen by a popular individual should add more to one’s popularity. Being nominated as powerful by a powerful one should contribute more to one’s perceived power. In the context of a communication network, those who are receiving many communications from others will themselves be valuable sources of information.

If one’s influence domain is full of prestigious nodes, one’s prestige should also be high. If, however, a node’s domain contains only peripheral or marginally important nodes, then the rank of this node should be low. Thus, we should weight the distances used in the proximity indices by considering the node importance in the influence domain.
• The NodeRank of a node is proportional to the sum of the NodeRanks of all nodes to which this node is directly connected.

\[ r_i = \lambda \sum_{j=1}^{n} a_{ij} r_j \]

Where \( r_i \) is the NodeRank of node \( i \), and \( a_{ij} \) = 1 if node \( i \) and node \( j \) are adjacent, or 0 otherwise. Hence, a node that is connected to many well-connected nodes is assigned a high score by this measure. However, a node that is connected only to near isolate nodes is assigned a low score, even if it has a high degree.

On the other hand, some aspects of nodes do not completely depend on their connection to others. For example, each student in a class has some popularity that is due to his or her external status characteristics. In a communication network, each individual has sources of information that are independent of other group members.

• A node has its own internal importance, which are independent of those of other nodes.

On the whole, a measure of the importance of a node should contain two factors: both external and internal. The above equation can thus be modified:

\[ r_i = \lambda \sum_{j=1}^{n} a_{ij} r_j + e_i \]

where \( e_i \) is the internal importance score of node \( i \). Moreover, these \( n \) equations for all nodes can be rewritten in a matrix form. Let \( r \) be a vector of Node Importance Scores, i.e. \( r = (r_1, r_2, ..., r_n)^T \), \( e \) be a vector of the internal importance of nodes, i.e. \( e = (e_1, e_2, ..., e_n)^T \), and \( A \) be an adjacent matrix\(^1\), then we have the following equation with a combination of both external and internal factors:

\[ r = \lambda A r + e \]

The parameter \( \lambda \) reflects the relative importance of exogenous versus endogenous factors in the determination of importance scores. This has the matrix solution:

\[ r = (I - \lambda A)^{-1} + e \]

where \( I \) is the identity matrix of dimension \( n \), and \( e \) is a vector of length \( n \).

\(^1\)An adjacency matrix of a graph with \( n \) nodes is an \( n \) by \( n \) square matrix: \( A = (a_{ij}) \), where \( a_{ij} = 1 \), if node \( i \) and node \( j \) are adjacent, or 0 otherwise.
CHAPTER 4. NODERANK: FILTERING FOR LAYOUT

We normalize \( r \) to a length of 1, i.e.

\[
|r| = \sqrt{\sum_{i} r_i^2} = 1
\]

The eigenvector importance of a node \( u_i \) is finally attained:

\[
R(u) = r_i \quad (4.2)
\]

Namely, the components of vector \( r \) are the corresponding \textit{NodeRanks} of nodes in a graph. High rank importance scores imply that nodes are connected either by a few other nodes which have high rank scores, or by many others with low to moderate rank scores. Recall that a node’s rank is the weighted sum of the ranks of those who connect to the node.

Eigenvector centrality can be either understood as a variation of simple degree, or interpreted as a refined version of degree.

This idea of eigenvector centrality was anticipated by [Katz 1953] and further developed by [Hubbell 1965] and many others, finally culminated with [Bonacich 1972] who defined centrality as the principle eigenvector of the adjacency matrix.

4.3 Cutsets and Bridges

After ranking all the nodes in a graph according to their importance scores, we remove all the nodes whose \textit{NodeRanks} are under a threshold and their connected edges.

As mentioned before, the underlying purpose of filtering a graph is to reduce visual complexity. A filtered graph can still represent the main relationships between nodes after such a filtering. That is, the basic requirement is that a filtered graph should be connected. The following definitions and theorem from graph theory [Harary 1972] are useful in keeping a graph connected.

\textbf{Definition 4.5} A graph is connected if, for every pair of nodes \( v_1 \) and \( v_2 \), there is a path between nodes \( v_1 \) and \( v_2 \).
A graph is said to be connected if it can be traveled from any one node to any others by moving along paths of edges. The graph is 2-connected if deletion of any node still keeps it connected; it is 3-connected if it still remains connected with removal of any two nodes, and so on. It is required that a $k$-connected graph have at least $k + 1$ nodes.

**Definition 4.6 CutPoint:** A node is a cutpoint if its removal disconnects a graph, i.e. increases the number of components. Also, it makes some points unreachable from some others.

The concept of a cutpoint can be extended from a single node to a set of nodes necessary to keep a graph connected. Those nodes are referred to as a cutset. A node cutset is a subset of the nodes of a graph, whose removal (simultaneously removing all edges adjacent to those nodes) makes the graph no longer connected. If the set is of size $k$, then it is called a $k$-node cut, denoted by $k(G)$. That is, the $|k(G)|$ of a graph is the minimum number of nodes that must be removed to make the graph $G$ disconnected.

For example, node $N_5$ in Figure 4.2 (a) is a cutpoint, i.e. 1-node cut, or $1(G)$, because its deletion creates two components in Figure 4.2(b).

![Figure 4.2: An example of Cutpoint and Bridge](image)

**Definition 4.7 Bridge:** An edge is a bridge if its removal results in disconnected subgraphs. Edge $(N_3, N_5)$ is a bridge because its deletion creates two components, as shown in Figure 4.2(c).

A bridge is an edge such that the graph containing the edge has fewer components than the sub-graph that is obtained after the edge is removed.
Denoting the bridge set as $\lambda(G)$, the edge connectivity $|\lambda(G)|$ of a graph is the minimum number of edges that must be removed to disconnect $G$.

**Theorem 4.1**

1. A node $v$ is a cutpoint if and only if there exists nodes $x$ and $y$ such that $v$ is on every path connecting $x$ and $y$.
2. Edge $e$ is a bridge if and only if there exists nodes $a$ and $b$ such that $e$ is on every path connecting $a$ and $b$.
3. Edge $e$ is a bridge if and only if $e$ is not contained in a circuit.

A complete graph has no cutpoint. All nodes are adjacent to all others so that the removal of any one node would still leave the graph connected.

As mentioned before, the basic prerequisite of filtering is that a filtered graph can still represent the main relationships between objects in the original graph. Therefore, those nodes and edges representing relatively complex relationships should remain after filtering.

Formally, let $F$ be a set of filtered nodes and associated edges, $t$ a threshold of NodeRank, and $G'$ the filtered graph. We have the following properties of a filtered graph as a summary:

1. $G' = G - F$
2. $F = \{(v \mid v \in V \land v \notin k(G) \land R(v) < t\}, \{(v, u) \mid u \in V \land (v, u) \notin \lambda(G)\})$
3. $|G'| \leq |G|$
4. $G'$ is connected

where $k(G)$ denotes the cutset of a graph $G$, $\lambda(G)$ denotes the bridge set, and $R(v)$ is the NodeRank of node $v$. 
CHAPTER 4. NODERANK: FILTERING FOR LAYOUT

4.4 Filtering Graphs

There are two filtering models: global filtering and fisheye filtering, both of which rely on thresholds. However, global filtering permanently removes relatively unimportant nodes (and their associated edges), measured by their NodeRanks. In other words, the appearance of a node is mainly determined by its NodeRanks. In the Fisheye filtering model, whether a node should be visible or not is conditional on both the NodeRanks of this node and its distance from the current focus node.

4.4.1 Global Filtering

With the ranked nodes, a filtered graph can be produced by suppressing all the nodes whose NodeRanks are less than a threshold, which is specified and adjusted by users.

Suppose that we denote the threshold as $t$, and then a set of the remaining nodes after filtering is

$$V' = \{ u \mid u \in V \land R(u) \geq t \}$$

where $R(u)$ is the NodeRank of node $u$.

An alternative means of obtaining the threshold is to specify what proportion of nodes will be reserved after filtering:

$$\frac{|V'|}{|V|} \leq t$$

The procedure is as follows: rank the nodes in decreasing order of their NodeRanks, and then sequentially remove the number of $(1-t)|V|$ nodes in order by starting from a node with the smallest score.

In order to measure the alteration of a graph after filtering, the density of a graph [Faust 1995] is employed:

$$\nabla(G) = \frac{2|E|}{|V||V|-1} = \frac{\overline{\text{Deg}}(G)}{|V|-1}$$

where $\overline{\text{Deg}}(G) = 2|E|/|V|$ is the average degree of the nodes in the graph.
This indicates the proportion of possible edges that are actually present in a graph. In other words, the density of a graph is the average proportion of edges incident with nodes in the graph.

The additional properties of a filtered graph \( G' \) are as follows:

- \(|V'| \leq |V| \) and \(|E'| \leq |E| \)
- \(\nabla(G') \geq \nabla(G)\)

A filtered graph not only reduces visual complexity, but also still captures the main relationships between the objects in an original graph.

### 4.4.2 Fisheye View

Furnas [Furnas 1981; Furnas 1986] devised the fisheye lens model as an efficient way of showing large graphs. This model uses a “degree of interest” function to constrain (called Filtering fisheye views, FEV) or distort the display of information to relevant or interesting elements, according to their importance. The degree of interest (DOI) of a node \( x \) is a function of a priori importance (API) of node \( x \) and the distance (D) between node \( x \) and focus node \( f \) [Furnas 1981, Furnas 1986]:

\[
DOI(x) = API(x) - D(x, f)
\]

It is clear that the degree of interest of a node increases with API and decreases with the distance. Generally, the importance and distance factors can be related to size, level in the hierarchy, or some other characteristics of objects in the collection. In the context of filtering a graph, \( API(x) \) is naturally determined by the importance of a node \( x \), which is the exact implication of Node Importance Score. Hence we have:

\[
API(u) = R(u)
\]

As for the distance, \( D(x, f) \) is the graph-theoretic distance between a node \( x \) and the focus node \( v_f \), i.e., \( D(u, f) = d(u, v_f) \). Suppose \( D \) denotes the maximum distance over all pairs of nodes which is the diameter of the graph \( D = \max_{u, v_f \in V} d(u, v_f) \). The degree of interest can therefore be rewritten as:

\[
DOI(u) = API(u) - D(u, v_f) \\
= R(u) - d(u, v_f) / D
\]
With this normalization, the value of $\text{DOI}(u)$ ranges within $(-1, 1]$. Given a node $u$, if $\text{DOI}(u) \geq t$, then the node $u$ will be visible, i.e. $\{u \mid u \in V \land \text{DOI}(u) \geq t\}$; otherwise invisible in the filtered graph, where $t$ is the threshold. The effect of the filtering fisheye view is that nodes in the neighborhood areas of the focus node are shown in detail, while only the most important nodes in more distant areas are displayed, as determined by the above equation.

4.5 Algorithms

The algorithm for filtering a graph is as follows:

---

**Input:** A connected graph, and a threshold

**Output:** A connected and filtered graph

1. Compute the NodeRank of every node in a graph
2. Rank the nodes by their NodeRanks
3. Remove nodes not being Cutpoints, whose NodeRanks are below the threshold, as well as associated edges that are not Bridges.

---

The above algorithm relies on several basic algorithms that we review as follows:

A **graph is connected:** A simple depth-first or breadth-first search suffices to test whether a graph is connected and to identify all the components in linear time. Specially, the simplest algorithms for identifying cutpoints (or bridges) would try to delete nodes (or edges) one by one, and then to use Depth-First Search (DFS) or Breadth-First Search (BFS) in order to test whether the resulting graph is still connected. More complicated but linear-time algorithms exist for both problems, based on a depth-first search.

The **shortest path between two nodes:** There are two main algorithms for finding the shortest paths in a graph: Floyd's Algorithm and Dijkstra's Algorithm.

**Eigenvector centrality:** This can be computed as follows [Hotelling 1936; Loan 1996]:

**Step 0:** Set $r_i=1$, and assign initial values to $e_i$ and $\dot{\lambda}$, for all node $i$

**Step 1:** Compute $r_i^* = \dot{\lambda} \sum_j a_{ij} r_j + e_i$
Step 2: Set $\gamma$ equal to the square root of the sum of squares of $r^*$ (the vector)

Step 3: Set $r_i = r_i^* / \gamma$ for all nodes $i$

Step 4: Repeat steps 1 to 3 until $\gamma$ stops changing

Note that after executing Step 1 the first time, $r^*$ is equal to a simple degree. The running time for this computation comes mainly from Step 1, i.e. $O(n^2)$. For a large graph, we can find the linkage of the graph offline, or gradually obtain part of it on-the-fly and then filter it.

### 4.6 Filtering Examples

In this section we present several examples. The first simple example is a random graph with 17 nodes and a density of 0.0735 shown in Figure 4.3(a). Figure 4.3(b) shows the filtered version of this graph, with 13 remaining nodes and a density of 0.0879. Although the number of nodes in the filtered graph decreases from 17 to 13 due to the increased threshold for controlling the appearance of nodes, its density increases by 0.0144. This demonstrates that the filtered graph can preserve the main relationships represented in the original graph.
Several application examples will be discussed below.

### 4.6.1 A Small Document Collection on “recipe”

In this example, we analyze the structure of a small document collection on *recipe* using NodeRank. Web pages and hyperlinks of this collection were gathered using the Web crawling software named *WebCrawler*, which was described in Chapter 3. The resulting layouts are shown in Figure 4.4, where nodes represent Web page documents, and edges indicate hyperlinks among the documents.

Table 4.1 lists the centrality indices and *NodeRanks* of the documents. The second, third, and fourth columns of this table show the *Degree*($C_D$), *Closeness*($C_c$) and *Betweenness Centrality*($C_B$) indices of the nodes, respectively. The remaining two columns in the table are *NodeRanks*, $R_n$ and $R(i)$, calculated by (4.1) and (4.2). From Table 4.1, compared with the other nodes, *Recipes* node (node 9) is obviously a prominent node with respect to those measures. Also, note that a different centrality index may lead to diverse interpretations. Node 1, for example, has the same value of *Closeness centrality* as that of node 11, but they are quite different in their *Betweenness* centralities. As previously mentioned, different centrality measures focus on various aspects of the structure of a graph. It is for this reason that we use their combination in
the equation (4.1) as the computation of NodeRanks, instead of a single centrality measure.

Roughly, there are two kinds of documents in the collection: “hub” documents with many links, and “sink” documents with incoming links, but without out-going links [Kleinberg 1999; Page 1998]. In the Figure 4.4, for example, Recipes (node 9) and Japanese Fried Rice (node 7) are “Hub” documents while Numerical Recipes (node 1), Oatmeal Cookies (node 6), Eggs Pepper (node 8) and Main Dishes (node 10) are “sink” documents. The importance of “Hub” documents surpasses that of other documents so that they have relatively high NodeRanks.

Figure 4.4: The structure of a small document collection on Recipes
A “hub” is a transitional document through which users move to certain destinations, while a “sink” tends to be a final destination.

Figure 4.5 illustrates the centrality indices and importance values of documents in the collection.

<table>
<thead>
<tr>
<th>Node</th>
<th>$C_D$</th>
<th>$C_c$</th>
<th>$C_B$</th>
<th>$R_n$</th>
<th>$R(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.071</td>
<td>0.368</td>
<td>0.000</td>
<td>0.146</td>
<td>0.1869</td>
</tr>
<tr>
<td>2</td>
<td>0.214</td>
<td>0.483</td>
<td>0.212</td>
<td>0.303</td>
<td>0.3450</td>
</tr>
<tr>
<td>3</td>
<td>0.143</td>
<td>0.438</td>
<td>0.093</td>
<td>0.225</td>
<td>0.2994</td>
</tr>
<tr>
<td>4</td>
<td>0.214</td>
<td>0.400</td>
<td>0.104</td>
<td>0.239</td>
<td>0.4000</td>
</tr>
<tr>
<td>5</td>
<td>0.214</td>
<td>0.389</td>
<td>0.055</td>
<td>0.219</td>
<td>0.3663</td>
</tr>
<tr>
<td>6</td>
<td>0.071</td>
<td>0.333</td>
<td>0.000</td>
<td>0.135</td>
<td>0.1060</td>
</tr>
<tr>
<td>7</td>
<td>0.286</td>
<td>0.467</td>
<td>0.255</td>
<td>0.336</td>
<td>0.4089</td>
</tr>
<tr>
<td>8</td>
<td>0.071</td>
<td>0.326</td>
<td>0.000</td>
<td>0.132</td>
<td>0.1256</td>
</tr>
<tr>
<td>9</td>
<td>0.429</td>
<td>0.560</td>
<td>0.522</td>
<td>0.504</td>
<td>0.6086</td>
</tr>
<tr>
<td>10</td>
<td>0.071</td>
<td>0.286</td>
<td>0.000</td>
<td>0.119</td>
<td>0.0634</td>
</tr>
<tr>
<td>11</td>
<td>0.214</td>
<td>0.368</td>
<td>0.022</td>
<td>0.201</td>
<td>0.3900</td>
</tr>
<tr>
<td>12</td>
<td>0.214</td>
<td>0.500</td>
<td>0.114</td>
<td>0.276</td>
<td>0.4830</td>
</tr>
<tr>
<td>13</td>
<td>0.143</td>
<td>0.389</td>
<td>0.143</td>
<td>0.225</td>
<td>0.2064</td>
</tr>
<tr>
<td>14</td>
<td>0.286</td>
<td>0.438</td>
<td>0.103</td>
<td>0.276</td>
<td>0.5034</td>
</tr>
<tr>
<td>15</td>
<td>0.214</td>
<td>0.483</td>
<td>0.092</td>
<td>0.263</td>
<td>0.4608</td>
</tr>
</tbody>
</table>

Table 4.1: The Centrality indices of documents in the collection

---

$R_n$ is calculated by formula (4.1)
4.6.2 Web Graphs

Three examples of filtering Web graphs will be provided in this section. In the first example, Huang’s example [Huang et al. 2000] in Figure 4.1 is reconstructed. This graph and its filtered graphs are shown in Figures 4.7 (a), (b) and (c). The NodeRanks of this graph are reported in Table 4.2 and illustrated in Figure 4.6. The numbers with asterisks in the last column of Table 4.2 are less than the threshold so that their corresponding nodes were removed as shown in Figure 4.7(b). From this table, we reach the following conclusions: (1) the DCB centrality is consistent with the Eigenvector centrality for most nodes (see Figure 4.6). Therefore we can use either of these two measures to achieve a filtered graph; (2) the nodes “Dept.html” and “logo.gif” have the highest centralities. This conforms to our observation in that such two nodes have highest degrees and locate at the most central positions of the graph; (3) the marginal nodes have lower NodeRanks such as nodes “w3.gif,””pic.jpg”,”w2.gif”, and “w1.gif”; (4) those nodes who have the similar connections and locate at the same positions in a graph have almost equal NodeRanks. Note that there are several groups of nodes with equal NodeRanks, such as nodes “w12.gif”, “force.class”,”w7.gif”, and “w13.gif”. This means that they cannot be distinguished from each other in terms of both measures; (5) in some cases, those nodes locating at the same positions in a graph, however, have different NodeRanks. For example, node “w12.gif” has a NodeRank of 10.76 while node “w13.gif” has a NodeRank of 11.43. If we look them carefully, they actually have various Closeness centralities with 30.25 and 32.24, respectively, which lead to their different NodeRanks.

The two additional examples (Figure 4.8 and Figure 4.9) were obtained as follows: first, use WebCrawler with two starting URL addresses: http://www.it.swin.edu.au, http://www.unimelb.edu.au, and search depth 4, to produce a file which includes all the URLs of the hyperlinked Web pages in part of the above Web sites, respectively. From this data collection, Web graphs are constructed where nodes represent Web pages, and edges represent hyperlinks among these pages. We then apply our proposed algorithm to filtering such Web graphs. Finally, the filtered graphs are laid out.

From these examples, we can conclude that: (1) the increase of the threshold results in the removal of more nodes and their associated edges. Equally well, the remaining
nodes are those nodes with relatively high NodeRanks; or those that are adjacent to prominent nodes; (2) the density of a filtered graphs increases as the number of their nodes decreases.

Figure 4.6: Comparisons of Node Importance Values by DCB\(^3\) and Eigenvector\(^4\) in Table 4.2

---

\(^3\) DCB is calculated by formula (4.1)  
\(^4\) Eigenvector is calculated by formula (4.2), with \(e = (0,0,\ldots,0)\)
### Table 4.2: The centrality indices of nodes in a Web graph of Figure 4.7

<table>
<thead>
<tr>
<th>Node</th>
<th>Node name</th>
<th>Degree</th>
<th>Closeness</th>
<th>Betweenness</th>
<th>DCB</th>
<th>Eigenvector</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Seminar.html</td>
<td>12.245</td>
<td>41.525</td>
<td>15.816</td>
<td>23.19533</td>
<td>23.354</td>
</tr>
<tr>
<td>2</td>
<td>w3.gif</td>
<td>2.041</td>
<td>29.518</td>
<td>0.000</td>
<td>10.51967</td>
<td>3.874*</td>
</tr>
<tr>
<td>3</td>
<td>pic.jpg</td>
<td>2.041</td>
<td>29.518</td>
<td>0.000</td>
<td>10.51967</td>
<td>3.874*</td>
</tr>
<tr>
<td>4</td>
<td>w2.gif</td>
<td>2.041</td>
<td>29.518</td>
<td>0.000</td>
<td>10.51967</td>
<td>3.874*</td>
</tr>
<tr>
<td>5</td>
<td>w1.gif</td>
<td>2.041</td>
<td>29.518</td>
<td>0.000</td>
<td>10.51967</td>
<td>3.874*</td>
</tr>
<tr>
<td>6</td>
<td>logo.gif</td>
<td>22.449</td>
<td>55.682</td>
<td>38.757</td>
<td>38.96267</td>
<td>60.301</td>
</tr>
<tr>
<td>7</td>
<td>Dept.html</td>
<td>32.653</td>
<td>52.688</td>
<td>36.277</td>
<td>40.53933</td>
<td>64.998</td>
</tr>
<tr>
<td>8</td>
<td>Postgrad.html</td>
<td>18.367</td>
<td>43.363</td>
<td>17.035</td>
<td>26.255</td>
<td>32.613</td>
</tr>
<tr>
<td>9</td>
<td>w22.gif</td>
<td>2.041</td>
<td>30.435</td>
<td>0.000</td>
<td>10.82533</td>
<td>5.410*</td>
</tr>
<tr>
<td>10</td>
<td>w9.gif</td>
<td>2.041</td>
<td>30.435</td>
<td>0.000</td>
<td>10.82533</td>
<td>5.410*</td>
</tr>
<tr>
<td>11</td>
<td>pic3.jpg</td>
<td>2.041</td>
<td>30.435</td>
<td>0.000</td>
<td>10.82533</td>
<td>5.410*</td>
</tr>
<tr>
<td>12</td>
<td>w11.gif</td>
<td>4.082</td>
<td>39.200</td>
<td>0.680</td>
<td>14.654</td>
<td>16.191</td>
</tr>
<tr>
<td>13</td>
<td>w10.gif</td>
<td>2.041</td>
<td>30.435</td>
<td>0.000</td>
<td>10.82533</td>
<td>5.410*</td>
</tr>
<tr>
<td>14</td>
<td>img2.gif</td>
<td>6.122</td>
<td>40.496</td>
<td>0.680</td>
<td>15.766</td>
<td>26.194</td>
</tr>
<tr>
<td>15</td>
<td>Staff.html</td>
<td>20.408</td>
<td>47.115</td>
<td>17.078</td>
<td>28.20033</td>
<td>45.056</td>
</tr>
<tr>
<td>16</td>
<td>Academic.html</td>
<td>16.327</td>
<td>42.982</td>
<td>17.786</td>
<td>25.69833</td>
<td>27.234</td>
</tr>
<tr>
<td>18</td>
<td>pic1.jpg</td>
<td>4.082</td>
<td>31.818</td>
<td>0.510</td>
<td>12.13667</td>
<td>9.489</td>
</tr>
<tr>
<td>19</td>
<td>w20.gif</td>
<td>2.041</td>
<td>29.878</td>
<td>0.000</td>
<td>10.63967</td>
<td>4.167*</td>
</tr>
<tr>
<td>20</td>
<td>w5.gif</td>
<td>2.041</td>
<td>29.878</td>
<td>0.000</td>
<td>10.63967</td>
<td>4.167*</td>
</tr>
<tr>
<td>21</td>
<td>w4.gif</td>
<td>2.041</td>
<td>29.878</td>
<td>0.000</td>
<td>10.63967</td>
<td>4.167*</td>
</tr>
<tr>
<td>22</td>
<td>up.class</td>
<td>2.041</td>
<td>29.878</td>
<td>0.000</td>
<td>10.63967</td>
<td>4.167*</td>
</tr>
<tr>
<td>23</td>
<td>Deport.html</td>
<td>2.041</td>
<td>34.752</td>
<td>0.000</td>
<td>12.26433</td>
<td>10.782</td>
</tr>
<tr>
<td>24</td>
<td>w.gif</td>
<td>2.041</td>
<td>34.752</td>
<td>0.000</td>
<td>12.26433</td>
<td>10.782</td>
</tr>
<tr>
<td>25</td>
<td>Awards.html</td>
<td>4.082</td>
<td>35.252</td>
<td>0.872</td>
<td>13.402</td>
<td>12.361</td>
</tr>
<tr>
<td>26</td>
<td>w15.gif</td>
<td>4.082</td>
<td>39.837</td>
<td>0.000</td>
<td>14.63967</td>
<td>18.255</td>
</tr>
<tr>
<td>28</td>
<td>imag5.gif</td>
<td>2.041</td>
<td>34.752</td>
<td>0.000</td>
<td>12.26433</td>
<td>10.782</td>
</tr>
<tr>
<td>29</td>
<td>Graph.html</td>
<td>10.204</td>
<td>41.176</td>
<td>8.078</td>
<td>19.81933</td>
<td>28.417</td>
</tr>
<tr>
<td>30</td>
<td>w16.gif</td>
<td>4.082</td>
<td>39.837</td>
<td>0.000</td>
<td>14.63967</td>
<td>18.255</td>
</tr>
<tr>
<td>31</td>
<td>Course.html</td>
<td>14.286</td>
<td>42.241</td>
<td>11.661</td>
<td>22.72933</td>
<td>32.082</td>
</tr>
<tr>
<td>33</td>
<td>w30.gif</td>
<td>2.041</td>
<td>29.878</td>
<td>0.000</td>
<td>10.63967</td>
<td>5.322*</td>
</tr>
<tr>
<td>34</td>
<td>w17.gif</td>
<td>2.041</td>
<td>29.878</td>
<td>0.000</td>
<td>10.63967</td>
<td>5.322*</td>
</tr>
<tr>
<td>35</td>
<td>w6.gif</td>
<td>4.082</td>
<td>31.210</td>
<td>0.000</td>
<td>11.764</td>
<td>11.391</td>
</tr>
<tr>
<td>36</td>
<td>w24.html</td>
<td>2.041</td>
<td>28.655</td>
<td>0.000</td>
<td>10.232</td>
<td>3.628*</td>
</tr>
<tr>
<td>37</td>
<td>Admin.html</td>
<td>4.082</td>
<td>33.108</td>
<td>0.000</td>
<td>12.39667</td>
<td>11.102</td>
</tr>
<tr>
<td>38</td>
<td>fram.class</td>
<td>2.041</td>
<td>28.655</td>
<td>0.000</td>
<td>10.232</td>
<td>3.628*</td>
</tr>
<tr>
<td>39</td>
<td>fun.class</td>
<td>4.082</td>
<td>31.613</td>
<td>0.425</td>
<td>12.04</td>
<td>8.146</td>
</tr>
<tr>
<td>40</td>
<td>w18.gif</td>
<td>2.041</td>
<td>29.341</td>
<td>0.000</td>
<td>10.46067</td>
<td>4.714*</td>
</tr>
<tr>
<td>41</td>
<td>img5.gif</td>
<td>2.041</td>
<td>29.341</td>
<td>0.000</td>
<td>10.46067</td>
<td>4.714*</td>
</tr>
<tr>
<td>42</td>
<td>w13.gif</td>
<td>2.041</td>
<td>32.237</td>
<td>0.000</td>
<td>11.426</td>
<td>7.474</td>
</tr>
<tr>
<td>43</td>
<td>ap.class</td>
<td>4.082</td>
<td>32.667</td>
<td>0.468</td>
<td>12.40567</td>
<td>9.524</td>
</tr>
<tr>
<td>44</td>
<td>w12.gif</td>
<td>2.041</td>
<td>30.247</td>
<td>0.000</td>
<td>10.76267</td>
<td>4.517*</td>
</tr>
<tr>
<td>45</td>
<td>force.class</td>
<td>2.041</td>
<td>30.247</td>
<td>0.000</td>
<td>10.76267</td>
<td>4.517*</td>
</tr>
<tr>
<td>46</td>
<td>w7.gif</td>
<td>2.041</td>
<td>30.247</td>
<td>0.000</td>
<td>10.76267</td>
<td>4.517*</td>
</tr>
<tr>
<td>47</td>
<td>w13.gif</td>
<td>2.041</td>
<td>30.247</td>
<td>0.000</td>
<td>10.76267</td>
<td>4.517*</td>
</tr>
<tr>
<td>48</td>
<td>Virtual.html</td>
<td>6.122</td>
<td>36.029</td>
<td>4.082</td>
<td>15.411</td>
<td>17.328</td>
</tr>
<tr>
<td>49</td>
<td>Monetary.html</td>
<td>2.041</td>
<td>30.435</td>
<td>0.000</td>
<td>10.82533</td>
<td>6.069</td>
</tr>
<tr>
<td>50</td>
<td>w14.gif</td>
<td>2.041</td>
<td>26.630</td>
<td>0.000</td>
<td>9.557</td>
<td>2.874*</td>
</tr>
</tbody>
</table>
(a) A graph with 50 nodes, NodeRank range\(^5\): 2.874\% - 64.998 \%, and Density 0.0380

\(^5\) The Node Importance Scores are calculated by Eigenvector, i.e., formula (4.2), which also applies to the following examples.
(b) A filtered graph with 33 nodes, NodeRank range: 5.322% ~ 64.998%, and Density 0.0720
(c) A filtered graph with 27 nodes, NodeRank range: 7.474%–64.998 %, and Density 0.1062

Figure 4.7: A Web graph
(a) A Web graph with 317 nodes NodeRank range: 0.018\%–37.438\%, and Density 0.0055
(b) A filtered Web graph with 261 nodes, \textit{NodeRank} range $^6$ : $2.086\% \sim 37.438\%$, and \textit{Density} 0.0070

Figure 4. 8: Part of Web graph of Web site of Swinburne University of Technology

---

$^6$ The threshold is 0.02
(a) A Web graph with 2000 Nodes, NodeRank range: 0.002%~27.549%, and Density 0.0016
Designing layout facilities for large graphs is one of the major challenges in the field of graph layout. Examples of large graphs include the World Wide Web, communication networks, knowledge bases, semantic networks, and so on. In this chapter, we have presented a novel method for filtering a graph. This method is motivated by measuring the importance role property of a node. The importance of a node is quantified as the degree to which it has direct and indirect relationships with others. In comparison with the rule-based approach, our approach is a structure-based one in that it makes use of the linkage of nodes rather than their semantics. As a consequence, it is applicable to any kinds of connected graphs. The illustrative examples have demonstrated that our approach can effectively reduce the number of nodes and edges, while the main linkage patterns in the original graph are still
reserved. It can be concluded that filtering is one efficient way to deal with large graphs. In the following two chapters, we will discuss an alternative way to resolve this problem.
Chapter 5

KNN: Clustering for Layout

Having presented the filtering approach to dealing with huge graphs in the last chapter, we now consider clustering as an alternatively efficient way. The first step in clustering is to develop a graph clustering algorithm. In this chapter and the next, two graph clustering approaches are presented. We start with formally defining the graph clustering problem, and then describe our first approach called KNN. In order to demonstrate the performance of our approach, the proposed approach is applied to clustering a Web graph. In Chapter 6, we continue the discussion of clustered graphs, but focus on the node similarity approach instead.

This chapter deals mainly with the issues in Clustering in the proposed framework in Figure 3.3. The content of this chapter has been published in [Huang and Lai 2003].

5.1 Introduction

Although many algorithms for graph drawing have been developed [Battista et al.1994; Battista et al.1999], most of these algorithms have difficulties in dealing with large graphs that have thousands of nodes. Clustering graphs serves as one efficient method to draw large graphs, apart from other techniques such as fisheye view, hyperbolic geometry [Lamping 1995], and distortion-oriented presentation [Keahey 1998]. A clustered graph can greatly reduce visual complexity by replacing a set of nodes in a cluster with an abstract node. Moreover, a hierarchically clustered graph can find superimposed structures over the original graph through a recursive clustering process.

Generally speaking, the purpose of cluster analysis is to organize data into meaningful groups: data in the same group are highly similar while those in different groups are dissimilar. Clustering is a process of finding such groups based on chosen
semantics. According to these semantics, the current clustering approaches can be roughly classified into two categories: content-based clustering and structure-based clustering. Content-based clustering uses semantic aspects of data such as category labels, while structure-based clustering takes advantage of structural information about data. Moreover, structure-based clustering is domain-independent so that it is suitable for graph visualization.

In order to cluster a graph, a metric of a node in the graph is required to quantify its features. Based on this metric, existing approaches of partitioning graphs [Everitt 1993; Hartigan 1975] can be loosely divided into the following groups: connectivity based partitions, which use standard concepts from graph theory such as components, cliques, and k-cores; distance partitions from selected subsets, which utilize neighborhoods of "central" nodes; neighborhood based partitions, in which a cluster is a set of units with similar neighborhoods such as degree partition, regular partition, and coloring; and other approaches, such as eigenvector methods. These approaches have already proven quite useful in graph partitioning. However, some of these approaches cannot be directly applied to graph visualization, because they depend on a good initial embedding of a graph, the creation of which is very expensive, particularly for a large graph.

In graph visualization, the node structural metric is widely utilized in many different forms. One simple example of such a metric is the degree of a node, i.e. the number of edges connected to the node. A metric more specific to trees, called the Strahler metric, is applied to tree graphs, in which nodes with the highest Strahler metric values generate a skeleton or backbone, which is then emphasized [Herman 1998; Herman 1999]. Using the distance metric, R.A. Botafogo et al. [Botafogo et al. 1992] constructed a distance matrix that has as its entries the distances of every node to every other node, to identify hierarchies in an organization.

A clustered graph can be laid out more quickly than the original graph as clustered nodes and edges significantly reduce graph complexity. The storage space of a clustered graph is, relatively speaking, less than that of the original graph. Furthermore, clustering nodes provide a basis for navigation and context clustering, which is accomplished by three possible approaches [Kimelman 1994]:

81
CHAPTER 5. KNN: CLUSTERING FOR LAYOUT

- Ghosting: de-emphasizing nodes, or relegating nodes to the background.
- Hiding: not displaying nodes with a metric under a cutoff value.
- Grouping: grouping nodes under a new supernode representation.

The Web graph has recently been used to model the link structure of the Web. The studies of such a graph can yield valuable insights into Web algorithms for crawling, searching and discovery of Web communities. This chapter will propose a new approach to clustering the Web graph. The proposed algorithm identifies a small subset of the graph as "core" members of clusters, and then incrementally constructs the clusters by a selection criterion. Two qualitative criteria are proposed to measure the quality of graph clustering. We have implemented our algorithm and tested a set of arbitrary graphs with good results. Applications of our approach include graph drawing and Web visualization.

The remainder of this chapter is organized as follows. We give the definitions and a formal description of graph clusters in the following section. Our approach is then described in detail in Section 5.3. In Section 5.4, we provide an empirical evaluation of a set of arbitrary graph clustering by using our approach. Applications are briefly depicted in Section 5.5. Finally, we summarize and conclude this chapter in Section 5.6.

5.2 Definitions

Suppose a given Web graph $G = (V, E)$ and an integer $k$, we partition $G$ into subgraphs $G_1, G_2, \ldots, G_k$, such that

$$G = \bigcup_{i=1}^{k} G_i \text{ and } G_i \cap G_j = \phi, i \neq j$$

An optimization criterion of this partition is known as the Minimum Cuts.

Given a subset $T$ of $V$, the cut $\delta(T)$ induced by $T$ is the subset of edges $(v_i, v_j) \in E$ such that $\{v_i, v_j\} \cap T \neq \emptyset$. That is, $\delta(T)$ consists of all those edges with exactly one endpoint in $T$. Actually, for each edge $e \in E$ if there is a nonnegative cost (or capacity) $c_e$, the cost of a cut $\delta(T)$ is then the sum of the costs of the edges in that cut:
\[ c(\delta(T)) = \sum_{e \in \delta(T)} c_e \]

The minimum cut is to find a cut with the minimum cost measured by the above equation. Since all the costs in our Web graph are 1s, the minimum cut thus becomes the problem of finding a cut with as few edges as possible in the graph.

A graph can be partitioned into different clusters in various ways so that we need a quantitative function to measure the quality of clustering. In general, a good clustering algorithm should produce clusters with both high “homogeneity (or cohesion)” and low “separation (or coupling)”. The nodes in a cluster are highly connected to others within one cluster, whereas the nodes in different clusters are separated with a minimum cut.

To measure the homogeneity of a cluster, we use the following function:

\[
\eta(C_i) = \begin{cases} 
\frac{2}{|C_i|(|C_i|-1)} \sum_{e_k \in C_i} |\{e_k\}| & \text{if } |C_i| \neq 1 \\
1 & \text{if } |C_i| = 1 
\end{cases}
\]

(5.1)

According to the above function, the cluster \( C_i \) will has \(|C_i|(|C_i|-1) / 2\) edges in total namely \( \eta(C_i) = 1 \), if the nodes in \( C_i \) are completely connected. In the case of a singleton cluster with \(|C_i| = 1\), we define the value of its homogeneity as 1.

**Definition 5.1 Homogeneity of clusters:** Give a graph \( G = (V, E) \) and its subgraphs \( G_1, G_2, \ldots, G_m \), then the average homogeneity of all the clusters of \( G \) is:

\[
C_H(G) = \frac{1}{|S|} \sum_{i} \eta(C_i)
\]

where \(|S|\) is the number of clusters in \( G \), \(|C_i|\) the number of nodes in the cluster \( C_i \), and \( \eta(C_i) \) given by equation (5.1).

Obviously the value of homogeneity ranges between \([0, 1]\). Observe that the bigger the value of \( C_H(G) \) is, the more cohesive the nodes in the clusters are.

In order to measure the degree of separation between two clusters, a function is required to quantify the relative costs of the clustering. It is clear that the sum of the
CHAPTER 5. KNN: CLUSTERING FOR LAYOUT

Edge costs in a minimum cut between two clusters is related to the sizes of the separated clusters. For this reason, a measurement of separation between two clusters can be given by:

\[
\delta(C_i, C_j) = \frac{c(\delta(T))}{\left| C_i \right| \left| C_j \right|} = \frac{1}{\left| C_i \right| \left| C_j \right|} \sum_{v_l, v_k \in C_i} c_{lk} \quad (5.2)
\]

where \( i \neq j \) and \( c_{lk} \) is the cost of the edge between nodes \( v_l \) and \( v_k \) with exactly one endpoint in the cut set. In our approach, we have \( c_{lk} = 1 \) if the nodes \( v_l \) and \( v_k \) are directly connected.

For a fully unconnected graph, namely without a cost for any cuts, we have \( \delta(C_i, C_j) = 0 \) from the above function. On the other hand, if clusters \( C_i \) and \( C_j \) are a complete bipartite graph where every pair of nodes in \( C_i \) and \( C_j \) are adjacent, then we have \( \delta(C_i, C_j) = 1 \).

**Definition 5.2** Separation of clusters: Given a graph \( G = (V, E) \) and its subgraphs \( G_1, \ldots, G_k \), we use the arithmetic average as the measurement of the separations among the clusters,

\[
C_s(G) = \frac{1}{n} \sum_{1 \leq i < j \neq j} \delta(C_i, C_j)
\]

where \( n \) is the number of pairs of clusters whose member node(s) are directly connected, \( 0 < n \leq k(k-1)/2 \), and \( \delta(C_i, C_j) \) is determined by equation (5.2).

Note that the smaller the value of \( C_s(G) \), the less the cost of the cut. With the above definitions, the graph clustering problem can be defined as follows.

**Definition 5.3** Maximum Homogeneity of Graph clustering: given a connected graph \( G = (V, E) \) and an integer \( m \), \( G \) is partitioned into subgraphs \( G_1, G_2, \ldots, G_k, \ldots, G_m \), in \( l \) different ways. If the following conditions hold true:

1. \( G = G^l = \bigcup_{i=1}^{m} G_i^l \)
2. \( G_i^l \cap G_j^l = \emptyset \) for \( i \neq j \)
CHAPTER 5. KNN: CLUSTERING FOR LAYOUT

(3) \( p = \arg \max \{ C_H(G^j) \} \)

(4) \( 0 \leq C_H(G^j) \leq 1 \)

Then \( G_1^p, G_2^p, \ldots, G_m^p \) are called Maximum Homogeneity Clusters of graph \( G \).

**Definition 5.4** Minimum Coupling of Graph Clustering: given a connected graph \( G = (V, E) \) and an integer \( m \). \( G \) is partitioned into subgraphs \( G_1^p, G_2^p, \ldots, G_m^p \), in the \( l \) different ways. If the following conditions hold true:

1. \( G = G_1^j = \bigcup_{i=1}^{m} G_i^j \)
2. \( G_i^j \cap G_j^i = \emptyset \), for \( i \neq j \)
3. \( p = \arg \min \{ C_S(G^j) \} \)
4. \( 0 \leq C_S(G^j) \leq 1 \)

Then \( G_1^p, G_2^p, \ldots, G_m^p \) are called Minimum Coupling of Graph Clusters of graph \( G \).

A few definitions used in the following algorithms are provided below.

**Definition 5.5** Node degree: The degree of a node \( v \) in a graph denoted by \( \text{deg}(v) \), is the number of edges incident with node \( v \). The sum of the degrees of \( G \) is \( 2|E| \), i.e.

\[
\sum_{i=1}^{n} \text{deg}(v_i) = 2 |E|
\]

since every edge has exactly two endpoints, the average degree of \( G \) is \( \mu = \overline{\text{Deg}}(G) = 2 |E| / l |V| \).

**Definition 5.6** Seed nodes: A set of nodes whose degrees are greater than \( \mu + \tau \), denoted by \( S \), where \( \tau \) is a threshold.

Let \( \sigma \) be standard deviation (a statistics term) of the degrees of a graph \( G \). The degree of a node \( v \) in a graph, denoted by \( \text{deg}(v) \), is the number of edges incident with node \( v \).

The threshold \( \tau \) is usually assigned as a value relating to \( \sigma \), for example, \( 0.1 \sigma \). The reason for this is that the standard deviation \( \sigma \) reflects the degree of the spread
of the node degree distribution. In the special case of a fully connected graph, namely the degree of every node is equal to $\mu$, there will be no seed node. This means that the clustering can start from any node in a graph.

In order to introduce the $k$-nearest neighbor search, we define a distance function as follows:

**Definition 5.7 Distance Function $D(v, u)$**: A function used to compute the distance between nodes $v$ and $u$ where $v$ and $u \in G$. In fact, it is the minimum length of all pairs of paths joining them, which is equal to the number of edges in the shortest path if such a path exists, otherwise $D(v, u) = \infty$. The shortest path can be found by using Dijkstra’s algorithm.

**Definition 5.8 $k$-nearest Neighbour Search**: For a query node $q \in G$ and a query parameter $k$, the $k$-nearest neighbour search returns a set $k$-NN $(q) \subseteq G$ that contains at least $k$ nodes in $G$, and the following conditions hold:

$$D(u, q) \leq D(w, q), \forall u \in k$-$\text{NN}(q), \forall w \in G - k$-$\text{NN}(q), u \neq w$$

Consequently, the resulting set is $\{w | w \in G - k$-$\text{NN}(q) \wedge D(w, q) \leq k\}$. For example, $1$-$\text{NN}(v)$ represents a set of nodes with which the node $v$ incidents. Note that the node $v$ is not included in the set of $k$-$\text{NN}(v)$.

**Definition 5.9 Affinity Function**: Given a connected graph $G = (V, E)$, the affinity of node $v$ to a cluster $C$ can be measured by:

$$f(v, C, k) = \frac{|C \cap k$-$\text{NN}(v)|}{\deg(v)} \quad \text{where} \quad v \notin C$$

Obviously, we have $0 \leq f(v, C, 1) \leq 1$

We will use these definitions to describe our algorithm in the following section.
5.3 Algorithm

The algorithm presented here has two steps: find the seed nodes (see Figure 5.1) and build a cluster around each such node recursively (see Figure 5.2). It initially detects, as previously defined seed nodes, those nodes whose degrees have greater than the average degree of the nodes in a graph. These nodes are potentially used as initial members of different clusters later. In some cases, two or more seed nodes are, however, densely connected and they are not far away in a graph. This suggests they should be within one cluster. Two seed nodes will thus be combined into one seed node, provided that they share the nodes of their $k$-NN, and the number of the shared nodes is no less than half a degree of one of them. Each member of the reduced seed node set is a core candidate for each cluster. After successively adding $1$-NN nodes of seed nodes into corresponding clusters, this algorithm allows each cluster to continually add new appropriate nodes from the $1$-NNs of each newly added node in the clusters and so on, until the affinity criterion of a node is not satisfied. Based on a defined affinity function, each cluster accepts a node as its new member. Such a function measures how the affinity of a node is relevant to existing member nodes in the cluster. In particular, the degree of a node affinity to a cluster, formally defined in Definition 9, is measured by the number of the shared nodes in both the $1$-NN of this node and the existing member nodes of this cluster. The clusters competitively choose a node as their new member according to the rule: the node with the highest affinity to a cluster. If the affinities of a node to all the clusters are equivalent, the node will become a singleton cluster. This procedure processes iteratively until no more nodes can be added to any cluster by following the above rule. Finally, the remaining nodes not belonging to any cluster must belong to chains. Such chains are then divided among their closest clusters, or remain as independent singleton clusters. The neighbouring nodes are found in an incremental manner in our algorithm. In other words, after having found the $k$ nearest neighbours, the algorithm does not make a computation from scratch to obtain the set of $k+1$ nearest neighbour, but continually explores the additional neighbours of current clustered nodes. Such an incremental nearest neighbour algorithm ranks the next available unclassified nodes in terms of their affinities to the newly classified members of the clusters, and each cluster then chooses the node with the highest affinity as its new member. The formal description of the algorithm is presented in Figures 5.1 and 5.2.
Input: graph $G = (V, E)$; seed set $S \leftarrow \emptyset$, threshold: $k$ and $\tau$

Output: $S$

Compute the degrees of nodes in $G$ denoted by $\text{deg}(v)$.
Compute the average degree $\mu$ and stand derivation of the degrees $\delta$

\[ \mu = \frac{2 |E|}{|V|} \]

//Find the seed nodes $S$

for each $v \in V$
    if $\text{deg}(v) \geq \mu + \tau$ then
        $S \leftarrow S \cup \{v\}$
    end if
end for

for $S = \emptyset$ then STOP

for each $v \in S$
    if $\max_{u \in S \setminus \{v\}} \left( (k \_ \text{NN}(v) + \{v\}) \cap (k \_ \text{NN}(u) + \{u\}) \right) \geq \text{deg}(u) / 2$ then
        $S \leftarrow S \setminus \{u\}$
    end if
end for

Figure 5.1: An algorithm to find Seed Nodes

// Construct the clusters around core candidate nodes

Input: seed set $S = \{s_1, s_2, \ldots, s_m\}$

Output: clusters $C_i (i = 1, 2, \ldots, m)$

for each $s_i \in S$
    If $1 \_ \text{NN}(s_i) \neq 1 \_ \text{NN}(s_j) (i \neq j = 1, \ldots, m)$ then
        $C_i \leftarrow C_i \cup \{s_i\} \cup 1 \_ \text{NN}(s_i)$
    else
        $k = \arg \max_{i=1, \ldots, m} \{f(1 \_ \text{NN}(s_i), C_i, 1)\}$
        $C_k \leftarrow C_k \cup \{s_i\} \cup 1 \_ \text{NN}(s_i)$
    end if
end for
for i = 1 to m
  for each  \( v \in V - C_i \)
    \( k = \arg \max_{i=1,\ldots,m} \{ f(v, C_i, l) \} \)
    if \( f(v, C_k, l) > 0.5 \) then
      \( C_k \leftarrow C_k \cup \{ v \} \)
    end if
  end for
end for

// This is an optional algorithm for handing the remaining node chain
for each  \( v \in V - C_i \)
  \( i = \arg \min_{i=1,\ldots,k} \{ D(v, u) \mid u \in C_i \} \)
  \( C_i \leftarrow C_i \cup \{ v \} \)
end if

Figure 5.2: Algorithms to aggregate nodes by local search

The algorithm can be summarized as follows:

1. Identify a small subset \( S \) whose members have high degrees in \( G \).
2. Remove the node(s) in \( S \) that is (are) highly connected to other node(s), and each remaining node in \( S \) is called a core candidate.
3. With a core candidate as its first member, each cluster incrementally classifies available nodes that have the highest affinities to the existing members of the cluster. That is, each member of each cluster continually absorbs the nodes in its \( 1-NN \) until the expansion criterion is not satisfied.

The time complexity of the algorithm is \( O(|S|^2) \).
5.4 Examples

We have implemented our algorithm in the prototype called PGD presented in Chapter 3. In this section, we present the clustering results of a set of arbitrary graphs by applying our approach, and discuss some features. Note that we will use one node to represent all the member nodes in a cluster in the following clustered figures.

The seed node set of Graph 1 in Figure 5.3 is \{v_4, v_7, v_9, v_{10}, v_{14}\}. The node v_9 has even affinities to other nodes v_4, v_7, v_{10} and v_{14}, and the node v_9 is thus a singleton cluster. It is actually also a bridge node between clusters \(v_{1-4}, v_{5-8}, v_{14-17}\) and \(v_{10-13}\). Note that all members in this seed node set are the core candidates for the corresponding clusters.

![Figure 5.3: Initial and Clustered Graph 1](image)

Graph 2 in Figure 5.4 was specially designed for testing our algorithm for dealing with a remaining node chain during clustering. First, the nodes \(v_1-v_5\) and \(v_6-v_{11}\) are constructed into two clusters, \(C_1(v_1-v_5)\) and \(C_2(v_6-v_{11})\). The node v_9 then aggregates one member of its 1-NN, node v_12, into cluster \(C_2\). That is, the node v_12 becomes a new member of \(C_2\). However, the node v_12 cannot accept node v_{13} into its cluster \(C_2\), because only one node of 1-NN (v_{13}) (node v_12) is within \(C_2\). This aggregation process thus stops at the node v_12 (See Figure 5.4(b)). In addition, the remaining node chain \(v_{13}-v_{16}\) can be optionally combined into \(C_2\) with the optional algorithm provided (See...
figure 5.4(c)). Or they just remain as different singleton clusters. This is reasonable in that the nodes in $C_1$ and $C_2$ are densely connected while the nodes $v_{13}-v_{16}$ are relatively not.

Graph 3 in Figure 5.5 shows various clustering results of a remaining node chain with different criteria. The clusters $C_1(v_1-v_3)$ and $C_2(v_4-v_8)$ are initially built. (Node $v_4$ has more affinity to $C_2$ ($v_5$ and $v_6$) than to $C_1(v_3)$ so that node $v_4$ will be put into cluster $C_2$). Then the remaining node chain $v_9-v_{11}$ is generated. With the loose criterion,

$$\max \{|1 - NN(v) \cap \{u | u \in C_2 \}| \geq \deg(v) / 2, \text{nodes } v_9-v_{11} \text{ will be incrementally aggregated into } C_2: v_9 \in 1_\_NN(v_6) \rightarrow v_{10} \in 1_\_NN(v_9) \rightarrow v_{11} \in 1_\_NN(v_{10}) \text{ (see Figure5.5 (b)).}$$

However, they will be singleton clusters with a strict criterion:

$$\max \{|1 - NN(v) \cap \{u | u \in C_2 \}| \geq \deg(v) / 2$$

The resulting clusters are shown in Figure 5.5 (c).
CHAPTER 5. KNN: CLUSTERING FOR LAYOUT

Figures 5.5: Initial and clustered Graph 3

Figures 5.6 and 5.7 show another two examples of well clustered Graphs 4 and 5 by our approach.

In our experiments, the parameters are $k=1$, i.e., $1$-NN and $\tau = 0$. If we want to reduce the number of clusters of a given graph, we should specify $\tau = \sigma$, for example. Then $\mu + \tau = 3.569$ will be used for clustering for Graph 3 shown in Figure 5.5. Only node $v_6$ (see Figure 5.5 (b)) can be a seed node based on Definition 5.6. This results in that nodes $v_1$, $v_2$ and $v_3$ will be singleton clusters with the loose criterion.
Chapter 5. KNN: Clustering for Layout

The experimental results for clustering are reported in Table 5.1. Some parameters are also illustrated in Figure 5.8. This shows our approach has fast running time and can find a good clustering solution.

5.5 Compare with Other Algorithms

The features of our approach differ from previous work in the following aspects:

1. Our approach does not require the initial layout of a graph and are suitable for clustering any types of graphs, including tree graphs, since it is based purely on the structure of a graph. Actually, the approach presented here is a novel method of graph partitioning so that it can be applicable to graph partition as well.

2. Without the need to be specified in advance, the number of clusters is automatically detected on the basis of the node degree distribution of a given graph—which differs from the $k$-means algorithm.

3. The number of clusters can be easily adjusted by changing the thresholds.

4. Within KNN, a proposed simple affinity function, as an expansion condition, is based on the degree of a node. The clusters are gradually expanded by local search with $k$-nearest neighbors. Actually, KNN identifies the number of clusters at a “global” level, and grows the nodes into the clusters by local search.
5. The definitions of homogeneity and separation are provided to formally measure the quality of graph clustering.

<table>
<thead>
<tr>
<th>Graph</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V</td>
<td>Deg(G)</td>
<td>σ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>3.294</td>
<td>0.470</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>3.875</td>
<td>1.668</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>2.363</td>
<td>1.206</td>
<td>2</td>
<td>0.863</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>3.500</td>
<td>0.978</td>
<td>5</td>
<td>0.511</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>3.778</td>
<td>0.808</td>
<td>3</td>
<td>0.873</td>
</tr>
</tbody>
</table>

Table 5.1: The parameters for clustering Graphs 1-5

5.6 Applications

The WWW can be considered as a graph where nodes are static html pages and edges are hyperlinks between these pages. This graph is called the Web graph. It has been the subject of a variety of recent work aimed at understanding the structure of the World Wide Web [Broder 2000; Albert 1999; Watts 1998; Dill 2001; Kleinberg 1999; Papadimitriou 2001; Kumar 2000; Kumar 1999].

The main findings about the WWW structure are as follows:
CHAPTER 5. KNN: CLUSTERING FOR LAYOUT

1. A power-law distribution of degrees [Albert 1999; Kumar 1999]: in-degree and out-degree distribution of the nodes of the Web Graph follows the power law. The probability that a Web page has in-degree $i$ is proportional to $1/i^x$, where the latest estimation for $x$ is 2.1. The out-degree of a Web page is also distributed with a power law with exponent roughly equal to 2.7. The average number of links per page is about 7.

2. A bow-tie shape [Broder 2000]: the Web's macroscopic structure, which can be naturally broken into four pieces: SCC, a Strongly Connected Component (a central core), where all the pages can reach one another along directed links; IN, consists of pages that can reach the SCC, but cannot be reached from it; OUT, consists of pages that are accessible from the SCC, but do not link back to it. Finally, TENDRILS, contains pages that cannot reach the SCC, and cannot be reached from the SCC.


4. Small world phenomenon [Watts 1998; Kleinberg 1999]: six degrees of separation between any two Web pages. It is almost true in a strongly connected component part of the Web graph if allowing traversal of both out-links and in-links, but not true in general.

5. Cyber-communities [Kumar 1999]: groups of individuals who share a common interest, together with the most popular Web pages among them. A bipartite clique in the Web graph can be interpreted as a core of such a community, defined by a set of fans, all pointing to a set of authorities, and the set of authorities all pointed to by the fans. The size of bipartite cliques is relatively small, ranging from 3 to 10. Over 100,000 such communities have been recognized in a sample of 200M pages on a Web crawl.

6. Self-similarity structure [Dill 2001]: the Web shows a fractal structure in many different ways. A graph can be viewed as the outcome of a number of similar and independent stochastic processes. At various scales, there are “cohesive collections” of Web pages (for example, pages on a site, or pages about a topic) and these
collections are structurally similar to the whole Web (i.e. they exhibit the *bow-tie* structure and follow the power-law of the in-degree and out-degree). The central regions of such collections are called “Thematically Unified Clusters” (TUCs), and they provide the navigational backbone of the Web.

Link analysis plays an import role in understanding of the Web structure. There are two well known algorithms for ranking pages: Page Rank [Page 1998] and HITS [Kleinberg 1999].

The study of the Web graph is not only fascinating in its own right, but also yields valuable insight into Web algorithms for crawling, searching and prediction of the Web structure. The Web graph is actually a special case of a general graph, and we can employ some existing graph theoretic approaches to classify the Web graph.

Our approach is based on the theoretical properties of a graph so that it potentially has wide applications. Here we apply it to Web visualization. As described in Chapter 3 in our prototype called PGD, there is a *WebCrawler* program that builds the graph of a Web site, with a starting address: http://www.it.swin.edu.au and a depth:5. The program applies the depth-first search algorithm to crawl the Web, and create a graph shown in Figure 5.9. Figure 5.10 shows the layout of its clustered graph applying the proposed algorithm, where white nodes are the member nodes of their black and nearby clustered nodes. Our approach actually identifies the dense subgraphs in the Web graph and then classifies them. From the clustered Web graph, we reach the following conclusions:

1. The clustered graph can reduce visual complexity. For large graphs, we will display only the clustered nodes and related edges. Although there are several existing approaches to layout large graphs, such as Hyperbolic tree [Munzner and Burchard 1995] and multilevel clustered graphs [Eades and Feng 1997], our approach has the feature of automatically clustering nodes based on the connectivity of a graph.

2. There is a possible Web community around the clustered node. We describe the properties of the Web communities derived as two distinct types of dense
subgraphs in Web graphs, where the link density is greater among members than between members and the rest of the network [Flake, Lawrence, Giles, Coetze 2002]. A Web community is a set of Web pages having a common topic. So far various graph theoretical approaches have been proposed to potentially extract Web communities from Web graph, such as HITS [Flake, Lawrence, Giles, Coetze 2002], bipartite graph [Kitsuregawa 2001] and the Maximum Flow algorithm [Lawrence, Flake, Giles 2002]. All these approaches are directly founded on the assumption that the dense part of Web graph is a potential Web community. We can identify a Web community by analysing the member nodes in the clusters in our approach.

3. Classification of Web pages. Hyperlinks contain high-quality semantic clues to the topic of a Web page. Pages on the same or related topics tend to be linked more frequently than those on unrelated topics [Chakrabarti 1999].
5.7 Summary

In this chapter, we have presented a new approach to clustering graphs. The approach produces good results in our experiments as well as when applied to the visualization of a Web site. Our approach is a graph theoretic one, and thus has wide applications. For example, if the input graph is a similarity graph, where nodes correspond to elements and edges connect elements with similarity values above some thresholds, it can be used to classify the elements. We will present an alternative way to cluster graphs in the next chapter.
Chapter 6

**SMK: Clustering for Layout**

In this chapter, we continue to address the problem of clustering graphs for their layouts, but focus on node similarity, which is another attempt to resolve this problem. KNN, described in the last chapter, is a gradual approach to utilising the connection of nodes for clustering. The proposed approach called SMK in this chapter takes advantage of similarities among nodes through their shared edges. Thus, the major difference between KNN and SMK is what kinds of node attributes are employed to measure the node connections in a graph. KNN identifies as clusters densely connected subgraphs directly using the links of nodes in a graph, whereas SMK encodes the linkage patterns of a graph into a similarity matrix, and then indirectly discover the corresponding subgraph clusters by grouping the most similar nodes in such a matrix. In the following, SMK will be presented and be illustrated with experimental examples.

This chapter deals mainly with the issues in *Clustering* in the proposed framework in Figure 3.3. The main content of this chapter has been published in [Huang and Lai 2003].

### 6.1 Introduction

This chapter presents a new approach for hierarchically clustering a graph. The key idea behind this approach is to use an abstract node to express a set of the most similar nodes in a graph. This is achieved by initially grouping similar nodes from a node similarity matrix that is constructed on the basis of a novel node similarity metric. Such each group potentially represents a set of highly connected nodes in the graph. Then these groups are individually replaced with abstract nodes to form a higher abstraction level with a reduced dimension graph. This clustered graph again produces a new coarse graph and the above procedure recursively iterates until the number of nodes in the clustered graph falls below a threshold.
The proposed approach relies on a new proposed metric for the measure of similarities among nodes in a graph, and on the well-known k-means algorithm with a new heuristic method.

The rest of this chapter is organized as follows. We present a node vector space model in the following section, and then construct a node similarity matrix in Section 6.3, followed by several definitions in Section 6.4. Section 6.5 presents our algorithms for clustering graphs. Experimental results are reported in Section 6.6, including an example of visualization of part of Java 1.4 class diagrams. Section 6.7 compares two approaches KNN and SMK. Related work is reviewed in Section 6.8, followed by a summary in the last section.

6.2 Node Vector Space Model

We suppose a graph denoted by \( G (V, E) \) with \( n \) nodes (i.e. \( n = |V| \)) and \( e \) links (i.e. \( e = |E| \)) where \( V \) is the set of nodes and \( E \) is the set of edges between the nodes. An edge-by-node matrix \( R \) of \( G (V, E) \) can be defined as \( R = (r_{ij})_{exn} \), each entry of which can be constructed as

\[
r_{ij} = \begin{cases} 
1 & \text{if node } v_j \text{ is incident with edge } e_i \\
0 & \text{otherwise}
\end{cases}
\]

For the graph in Figure 6.1, the matrix \( R \) is as follows:

\[
R = \begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\end{bmatrix}
\]

Figure 6.1: An example of a graph and its edge-by-node matrix

In the field of information retrieval, a vector is used to represent key words or a document in a collection [Michael et al. 1999]. Similarly, each node in a graph can be

\footnote{It is actually an incidence matrix of \( G \)}
CHAPTER 6. SMK: CLUSTERING FOR LAYOUT

encoded as a vector where each vector component reflects the appearance of a particular edge in the node. Conversely, each edge can also be encoded as an edge vector, and thus the component indicates the appearance of a particular node in that edge.

A vector representing a node is called a node vector. The node vectors for all the nodes in a graph are stored as the column space of matrix $R$. In other words, each column of $R$ represents a node, whilst each row characterizes an edge (Figure 6.1). As a consequence, a graph can be expressed as a matrix. Namely the $n$ node vectors may be thought of forming the $exn$ node matrix, denoting the $i$-th node vector of $R$ as $r_i$. If a graph $G$ is a large graph, two characteristics of $R$ are high-dimensionality and sparsity.

6.3 Node Similarity Matrix

In order to cluster a graph, a node metric should be defined to quantify an abstract feature, and then the clustering can be accomplished by assigning the nodes to a group according to their converted metric values. In pattern recognition and machine vision, graph edit distance [Bunke 1997; Horaud 1989] and maximum common subgraph [Horrad 1989] are used to match graphs, i.e. to determine the similarity of two graphs. There is, however, little literature on how to calculate similarities among the nodes within a single graph. In this chapter, a node structural metric is thus proposed to measure such similarities, making use of the number of shared edges. The degree of similarity between two nodes is partly determined by the number of edges between them. In particular, the greater the number of edges the two nodes share, the more similar they are. At the same time, the greater the number of edges they do not share, the less similar they are. To this end, a similarity measure function is needed. The measures that occur most in the literature are the dot product, Euclidean distance and the Jaccard Coefficient [Everitt 1993]. Among them, Jaccard Coefficient can be used to measure the degree of overlap, which is defined as:

$$sim(a, b) = \frac{N_j(a_j = b_j = 1)}{N_j(a_j = 1) + N_j(b_j = 1) - N_j(a_j = b_j = 1)}$$

where $a$ and $b$ are binary vectors, and $N$ represents a counting operator. For instance, the above numerator denotes the number of an attribute $j$ (i.e. edge) occurring in both $a$ and $b$. 

101
CHAPTER 6. SMK: CLUSTERING FOR LAYOUT

From the above definition, the similarity degree of two nodes can be calculated by their corresponding node vectors, as stated before. Recall that a node vector is a binary vector in which a component is only 1 or 0 to indicate whether or not an edge links to the node. Replacing a, b with node vectors in matrix $R$, the following equation is derived:

$$\text{sim}(r_i, r_j) = \frac{r_i^T r_j}{r_i^T r_i + r_j^T r_j}$$

$$= \frac{(R^T e_i)(e_j^T R)}{(R^T e_i)(e_i^T R) + (R^T e_j)(e_j^T R) - (R^T e_i)(e_j^T R)}$$

(6.1)

where $i$ and $j = 1, \ldots, n$, and $e_i$ (or $e_j$) denotes the $i$-th (or $j$-th) canonical vector of dimension $e$, i.e., $e = (1, 1, \ldots, 1)^T$.

It is quite reasonable to utilize the number of shared edges with regard to the measurement of node similarities if we look at the original purposes of what the nodes represent.

The use of the number of shared links finds a group of similar nodes in the sense that the connectivity or pattern of linkages between nodes contains a lot of implicit information about common features among the objects which the nodes represent. Namely a group of nodes containing analogous information has the likelihood of many links among them, while dissimilar nodes will have few or no links.

Consider, for example, the case of the Web graph where Web pages can be represented by the nodes and hyperlinks among those pages by the edges. It is possible that the Web pages with similar topics have more hyperlinks among them than those with different topics. Actually, there already exist two types of algorithms to extract the relationships between the nodes: PageRank [Page 1998] and HITS [Kleinberg 1999]. Both algorithms employ a bootstrapping approach to determining the quality or "authority" of a Web page using the number and quality of the pages that it links to. Conversely, from a graph perspective, the nodes in a highly connected subgraph are likely similar to each other in terms of what those nodes originally represent.

We return now to (6.1) with applying it to the graph shown in Figure 6.1. According to (6.1), for instance, the similarities between nodes $(v_1, v_4)$, and nodes $(v_1, v_2)$ are: $\text{sim}(r_1, r_4) = 0.167$, and $\text{sim}(r_1, r_2) = 0.200$, respectively. Note that the similarity value of nodes
CHAPTER 6. SMK: CLUSTERING FOR LAYOUT

$v_1$ and $v_4$ is less than that of nodes $v_1$ and $v_2$. The reason for this is that node $v_4$ contributes only one shared edge from its total four edges, although nodes $v_1$ and $v_2$ have one shared edge as nodes $v_1$ and $v_4$ do. Furthermore, we observe that the similarity values of all pairs of non-neighbour nodes are zero by formula (6.1): $\text{sim}(r_1, r_6) = 0$, for example. Clearly, this is not accurate in real applications. If node $v_1$ represents a paper with a reference paper $v_4$ that in turn refers to a paper $v_6$, then paper $v_1$ should be somewhat related to paper $v_6$. Using a transitive similarity can effectively solve this problem. In the above example, we have $\text{sim}(r_1, r_6) = \text{sim}(r_1, r_4) \ast \text{sim}(r_4, r_6)$ because an existing path from $v_1$ to $v_6$ is $(v_1, v_4)$ and $(v_4, v_6)$.

In general, it is possible for two non-neighbour nodes in a graph to have more than one or no paths between them. We aim to maximize the similarity degree between two nodes. Obviously, the longer the path between two nodes is, the less the similarity value they have. The maximum similarity degree of two nodes can consequently be considered as finding the shortest path with respect to the minimum cost in the graph where the cost of every edge is 1.

For the above reason, finding the shortest paths between all pairs of nodes in a graph necessitates the prerequisite step. More precisely, the shortest paths between two non-neighbour nodes, defined as a path with the fewest edges, are initially found by the well-known Dijkstra or Floyd’s algorithms. The products of sequentially multiplying similarity values of all node pairs in such paths are then calculated. Finally, the maximum value among those products is chosen as the degree of similarity between the two nodes. Formally, it is assumed that one of the shortest paths between nodes $v_i$ and $v_j$ is along a sequence of node pairs $(v_i, v_k), (v_k, v_l), \ldots, (v_m, v_j)$, and that their corresponding node vector sequence consist of a set of $P$: $(r_i, r_k), (r_k, r_l), \ldots, (r_m, r_j)$. The similarity value $s(r_i, r_j)$ maximizes all the values of $\text{sim}(r_i, r_j)$ over all the possible shortest path sets, denoted by a union set $P'$, between nodes $v_i$ and $v_j$. An equation is accordingly arrived at:

$$s(r_i, r_j) = \begin{cases} \max_{P \in P'} \left\{ \prod_{(r_t, r_k) \in P} \text{sim}(r_t, r_k) \right\} & \text{if } P \neq \emptyset \\ 0 & \text{if } P = \emptyset \end{cases}$$

(6.2)

where $1 \leq i, j \leq n$ and $i \neq j$. Note that the calculation of similarities between two neighbor nodes ($\text{sim}(r_i, r_k)$) can be treated as a special case of that of two non-neighbor nodes ($\text{sim}(r_i, r_j)$) in the above equation.
With combination of (6.1) and (6.2), the node similarity matrix of a graph $G$ can be constructed, where each entry is computed by formula (6.2) which tells the extent to which two nodes are similar. The node similarity matrix is thus derived as follows:

$$S = \left[s(r_i, r_j)\right]_{non}$$

The algorithm for constructing a similarity matrix for a given graph $G$ can be briefly summarized in Figure 6.2:

**Input:** $G (V, E)$  
**Output:** A node similarity matrix $S$

- Construct the edge-by-node incident matrix $R$ of the graph $G$
- Calculate similarity values of neighbor node pairs according to (6.1)
- Find the shortest paths between non-neighbor node pairs by using Dijkstra’s algorithm if they exist
- Construct the node similarity matrix according to (6.2)

**Figure 6.2: Algorithm for constructing a node similarity matrix**

In the case of the graph in Figure 6.1, its symmetric similarity matrix of the graph is shown in Table 6.1.

$$S = \begin{bmatrix}
1.000 \\
0.200 & 1.000 \\
0.200 & 0.200 & 1.000 \\
0.167 & 0.167 & 0.167 & 1.000 \\
0.007 & 0.007 & 0.007 & 0.042 & 1.000 \\
0.028 & 0.028 & 0.028 & 0.167 & 0.253 & 1.000 \\
0.007 & 0.007 & 0.007 & 0.042 & 0.333 & 0.250 & 1.000
\end{bmatrix}$$

**Table 6.1: The node similarity matrix of the Graph in Figure 6.1**

The formula (6.2) is in fact a metric to quantify the degree of similarity between two nodes with a graph, which satisfies fundamental properties: non-negativity, reflexitivity and symmetry.

**Theorem 6.1** For any two nodes $v_i$ and $v_j$ in a connected graph $G$, and their corresponding node vectors are $r_i$ and $r_j$, respectively, where $i \neq j$, the following properties hold true:
CHAPTER 6. SMK: CLUSTERING FOR LAYOUT

1. \( 0 \leq s(r_i, r_j) \leq 1 \)
2. \( s(r_i, r_i) = 1 \)
3. \( s(r_i, r_j) = s(r_j, r_i) \)

By means of the similarity matrix \( S \), we can easily obtain the dissimilarity matrix \( D \), which will be used as an input of the \( k \)-means algorithm in Section 6.5:

\[
D = [1]_{n \times n} - S
\]

The running time of this algorithm is \( O (e + n \log n) \).

6.4 Definitions

In this section we provide a few definitions, beginning with rephrasing the node vector used in the previous sections as Definition 6.1.

**Definition 6.1 Node Vector:** A vector denoted by \( r \in R^n \) representing a node with respect to the appearances of the edges in a given graph \( G = (V, E) \). The component of this vector is “1”, if the node is incident with an edge and “0” otherwise.

**Definition 6.2 Abstract Node:** A supernode or metanode representing a subgraph of the most similar nodes in a graph. The node vector of an abstract node is equivalent to the arithmetic mean of all node vectors of the nodes in such a subgraph.

Grouping the “similar” nodes into a number of clusters is achieved through maximising the mean of node similarity values. An abstract node represents the “cluster centre” or median of a cluster.

Suppose \( l \) denotes an abstract level of a clustered graph, i.e., \( l=0 \), the graph is the original graph without clustering. Also, suppose that \( e \) stands for the number of unique edges in the vector space model, and \( n \) is the total number of nodes in the \( l \)-th abstract level of the graph. Given the vector space model, the \( l \)-th abstract level (\( l \geq 1 \)) of node vectors can be represented by \( r_1', r_2', \ldots, r_n' \), where each \( r_i' \in R^e \). Clearly, the node vectors of the \((l+1)\)-th abstract level are directly derived from the node vectors of the \( l \)-th abstract
CHAPTER 6. SMK: CLUSTERING FOR LAYOUT

level. Let $\pi_{1}^{i-1}$, $\pi_{2}^{i-1}$, $\ldots$, $\pi_{n}^{i-1}$ denote a partitioning of the $l$-th abstract level of the node vectors into $n^{i-1}$ disjoint clusters, i.e.

$$
\bigcup_{i=1}^{n^{i-1}} \pi_{i}^{i-1} = \{r_{1}^{l}, r_{2}^{l}, \ldots, r_{n^{i-1}}^{l}\} \text{ and } \pi_{i}^{i-1} \cap \pi_{j}^{i-1} = \emptyset \text{ if } i \neq j
$$

For each fixed $1 \leq i \leq n^{i-1}$, the mean vector or the centroid of the node vectors contained in the cluster $\pi_{i}^{i+1}$ is

$$
m_{i}^{l} = \frac{1}{n_{i}^{l}} \sum_{r \in \pi_{i}^{i+1}} r
$$

where $n_{i}^{l}$ is the number of node vectors in the cluster. So we can get the $l+1$ th abstract level of the norm abstract node vector as

$$
r_{i}^{l+1} = \frac{m_{i}^{l}}{\|m_{i}^{l}\|}
$$

As a result, the $l+1$-th abstract level of the node vector set is $\bigcup_{i=1}^{n_{i}^{l+1}} r_{i}^{l+1}$.

In graph visualization, the focus is mainly on the similarities among a set of nodes, thus an object function is used to evaluate the “coherence” of clustering:

$$
Q = \sum_{j=1}^{n_{i}^{l+1}} \sum_{r_{i} \in \pi_{i}^{i+1}} \|r_{i} - r_{j}^{l+1}\|
$$

Intuitively, this objective function measures the combined coherence of all $n_{i}^{l+1}$ clusters for the $l$-th abstract level of a clustered graph. We attempt to minimize its value to achieve the best clustered results.

**Definition 6.3 Abstract Edge:** An aggregated edge that embodies existing links between any member nodes of a cluster and other nodes not belonging to this cluster. In other words, internal edges among members in a cluster will disappear while external edges, between cluster members and other outside nodes in the graph, will be contracted into abstract edges. Suppose $E_{v_{c}}^{l}$ denotes a set of the $l$-th abstract level edges of the abstract node $v_{c}$, then we have:

$$
E_{v_{c}}^{l} = \{e_{ij} \mid v_{i} \in C^{l} \land v_{j} \in G - C^{l}\}
$$

where $C^{l}$ denotes a set of nodes in a cluster that is represented by an abstract node $v_{c}$.
P. Eades et al. [Eades and Feng 1997] defined a hierarchically clustered graph as follows:

**Definition 6.4** [Eades et al. 1997] *Hierarchically Clustered Graph*: Denoted by $C = (G, T)$, consists of a graph $G$ and a rooted tree $T$ whose leaves are exactly the nodes of $G$. $T$ is called the inclusion tree, and represents a recursive inclusion relation called a clustering of $G$. Internal nodes of $T$ are called abstract nodes or clusters.

Obviously, a cluster represents a group of its children nodes and children clusters. Ultimately the cluster represents a group of both its children nodes, and recursively the children of its children clusters. For example, the root of $T$ represents a cluster that groups all the nodes of $G$. Since clustering is an inclusion relation, if two clusters contain a common node, then one of the clusters contains the other.

Any two clusters cannot contain a common node in our approach. The tree $T$ represents the abstract levels of a graph and the exclusion relations between parent and children clusters. The edges of a clustered graph in our approach are replaced with abstract edges that include only the edges among the nodes that are within different clusters.

Up to this point, a node similarity matrix and some definitions have already been presented. From this, it follows that a clustering algorithm is chosen in order to apply it to the matrix. The most important and challenging characteristics of the matrix are high dimensionality and sparsity, as mentioned before. For the purpose of efficiency, it is important that the chosen clustering algorithm should exploit the sparsity of the data while producing useful results at the same time. The $k$-means algorithm satisfies these requirements and hence is our choice of the algorithm.
6.5 Algorithms

From a matrix $R$ and a metric for quantifying node similarities, the node similarity matrix $S$ is obtained. After this, it is possible that clustering the nodes is based on such a matrix. Indeed, many clustering algorithms can accept a similarity matrix as an input. Here we choose the $k$-means algorithm since it is simple to program and easy to compute on large samples. There are, however, problems with this technique: specification of the number of clusters beforehand, and initial choices of centroids of clusters. In the following sections, an algorithm will be developed to efficiently identify the seed node set in a given graph. The number of members and each member node of this seed node set are equivalent to the number of clusters and the initial nodes of the cluster centroids, respectively. Then, the $k$-means algorithm, together with the seed node set as its initial input, is applied to the node similarity matrix. Finally, the main algorithm for hierarchically abstracting graphs is provided.

6.5.1 Determination of the Number of Clusters and Initial Nodes

A number of different schemes have been developed for selecting an initial set of seed nodes as the centroids of clusters [Hartigan 1975]. A commonly used scheme is to select the seeds at random. A clustering solution is then computed using each one of these sets. The quality of such clusters is evaluated by computing the similarity of each node to the centroid vector of the cluster that it belongs to. The best solution is the one that maximizes the sum of these similarities over the entire set of nodes.

Starting with arbitrary random centroids is, however, a relatively poor solution. An efficient method is presented to determine the number of clusters and then to choose initial centroids of the clusters.

The basic idea of this approach uses the Seed Nodes in Definition 5.4 in Chapter 5. Intuitively, a node with a relatively higher degree, i.e. more connectivities to other nodes, should form a local “community” together with the nodes around it.
CHAPTER 6. SMK: CLUSTERING FOR LAYOUT

The algorithm is as follows:

\[
\begin{aligned}
\text{Use the Figure 5.1 algorithm to find a Seed Nodes} \\
\text{if } |S| > N_{\text{max}} \text{ then} \\
\quad k_n = N_{\text{max}} \\
\text{else} \\
\quad k_n = |S| \\
\text{end if}
\end{aligned}
\]

Figure 6.3: An algorithm to find seed nodes

where \( S \) is the Seed Node set, and \( k_n \) is the number of clusters.

In the above algorithm, a node degree can be calculated by summing up the entries of the column in \( R \), \( i.e. \deg(v) = e^T r \). The maximum number of display nodes \( N_{\text{max}} \) is determined by the area of a display screen and the average occupied area of a node. The time complexity of this algorithm is \( O(|S|^2) \).

6.5.2 K-means

For the \( k \)-means algorithm, agglomerative clustering is employed that starts with the \( n \)-way partition and constructs iteratively a \( k \)-way partition from the \( k+1 \)-way partition. The algorithm consists of a simple re-estimation procedure as follows. First, the nodes in the \( \text{Seed Nodes} \) set \( S \) are individually assigned to the \( |S| \) cluster sets as their initial memberships. The centroid is then computed for each clustering set. Finally, the algorithm successively merges the available nodes into the nearest set. These steps are alternated until a stopping criterion is met, \( i.e. \) when there is no further change in the assignment of the nodes.

The \( k \)-means algorithm for abstracting the nodes:

---

**Inputs:**

- \( D \) (a node dissimilarity matrix to be clustered)
- \( S \) (the Seed Node set)

**Outputs:**

- \( C^n = \{c_1, c_2, \ldots, c_{|S|}\} \) (cluster centroids)
- \( m: D \to \{1 \ldots |S|\} \) (cluster membership)
Procedure $K$-Means

\[ C^n \leftarrow S \] (Set initial values)

\begin{verbatim}
for each \( d_i \in D \)
    \[ m(d_i) = \arg \min_{j=1,...,k} \text{distance}(d_i, C^j) \]
\end{verbatim}

while \( m \) has been changed

\begin{verbatim}
for each \( i \in \{1,...,|S|\} \)
    Recompute \( c_i \) as the centroid of \( \{d \mid m(d) = i\} \)
\end{verbatim}

\begin{verbatim}
for each \( d_i \in D \)
    \[ m(d_i) = \arg \min_{j=1,...,k} \text{distance}(d_i, C^j) \]
\end{verbatim}

distance (node \( v \), cluster \( C \)) {
    \begin{verbatim}
    for each \( c \in C \)
        \( (v, c) \rightarrow (i, j) \)
        \[ \text{sim}(v, c) = d_{ij} \]
    \end{verbatim}
    \[ distance = \sum_{c \in C} \text{sim}(v, c) \]
\}

---

**Figure 6.4:** The \( k \)-means algorithm

In the above algorithm, the distance between a node and a cluster centroid equals the sum of the dissimilarity values between this node and all the nodes in this cluster, which are the entries of the node dissimilarity matrix \( D \).

### 6.5.3 Main Algorithm

An algorithm for the hierarchical abstraction of a graph is described in Figure 6.5. The main algorithm begins with constructing the edge-by-node matrix \( R \) from a given graph \( G \), by expressing each node as the column and each edge as the row. Using the algorithm
in Figure 6.2 then derives the node dissimilarity matrix from this matrix. After that, the *Seed Nodes* set in each abstract level is identified by the algorithm in Figure 6.3. Next, applying the *k*-means algorithm to the easily derived dissimilarity matrix recursively produces the clustered graphs. Finally, a sequence of coarse graphs $C^l$ is laid out by the Spring Embedding algorithm [Eades 1984] in the next chapter. Overall, the algorithm is summarized as follows:

**Input:** $G = (V, E)$ and a threshold

**Outputs:** Multiple window layouts of clustered subgraphs of $G$

while $l < \text{threshold}$ or $|V| = 1$

Construct the node similarity matrix $S$ according to Fig. 6.2

$$D^l = [1]|_{|V|>|V|} - S$$

Find the *Seed Nodes* set $N$ according to Figure 6.3

Apply the *K*-means algorithm to $D^l$ with $S$

Replace all nodes in cluster memberships $m(D^l)$ with abstract nodes

Add abstract edges

Store the clustered graph to $C^l$

end while

Specify the abstract level $l$

MMD$^2$: Layout $C^l$

*Figure 6. 5: The Main Algorithm for Hierarchically Abstracting Graph*

**Theorem 6.2** The time complexity of the algorithm for hierarchically abstracting a graph is $O(|V|^2)$

**Proof:** The sketch of proof is as follows: The running time for constructing the similarity matrix is $O(|V|^2/2+|V|/2)$, because this matrix is symmetric. Using Dijkstra’s algorithm, the time complexity of finding the shortest paths for any two nodes is $O(|E| + |V| \log |V|)$. It takes $O(|V|^2/2)$ running time to calculate similarity values of node pairs. Finding the *Seed Nodes* set $N$ requires $O(|N|^2)$ computation where $|N|$ is the number of clusters. The time complexity of the *k*-means algorithm is $O(|V||N|l)$, where $|V|$ is the number of nodes, and $l$ is the number of iterations. The do-while loop is bounded by the value of the threshold, which is the number of the abstract levels. The threshold is 3 in our

---

$^2$ MMD will be described in the next chapter.
experiments. Therefore, the worst-case overall running time of the algorithm for hierarchically abstracting a graph is $O(|V|^2)$.

6.6 Examples

All the figures in this section are snapshots extracted from our experimental examples. As we will mention in the next chapter, different abstract views are displayed in various layer windows in our implementation. Note that the graphs in the following figures have been adjusted due to the limited page size.

A node set $\{v_1, v_2, v_3, v_4, v_6\}$ includes all the nodes in Graph 1 in Figure 6.6 (a), whose member degrees exceed the average degree. According to the algorithm in Figure 6.3, the Seed Nodes set in this instance should be reduced to $\{v_4, v_6\}$ in that the nodes $v_1$, $v_2$, $v_3$ and $v_4$ share at least three nodes of their $l_{NN}$ with each other. Figure 6.6 (b) shows the result after applying the $k$-means algorithm with the above Seed Nodes set as initial cluster members. The $\text{Open}(v_{1..4})$ operation Figure 6.6(b) results in the layout in Figure 6.6 (c), where node $v_{1..4}$ is an abstract node with the first abstract level.

![Diagram](image1)

(a) $l = 0, |V| = 7$

(b) $l = 1, |V| = 2$

(c) Result of the operation $\text{Open}(v_{1..i})$ in (b)

Figure 6.6: The layouts of hierarchically clustered graph of Graph 1 in Figure 6.1

In real applications, the layout sequence of a clustered graph starts from top to bottom in the abstract level tree. The display sequence of Graph 2, for example, is Figures 6.7 (c), (b) and (a). Figures 6.7(b) and (a) show the results of the operations of $\text{ExpandAll}(G^2)$ and $\text{ExpandAll}(G^3)$ defined in the next chapter, respectively. Alternatively, users can

---

3 This operation of MMD will also be introduced in the next chapter
interact with the layouts. As an example, the $\text{Open}(v_{18-21})$ operation reveals details of the abstract node $v_{18-21}$ in Figure 6.7(b).

![Diagram](image)

(a) $l = 0, |V| = 38$

(b) $l = 1, |V| = 6$

(c) $l = 2, |V| = 3$

Figure 6.7: The layouts of hierarchically clustered graph of the Graph 2

The nodes in the graphs in Figures 6.8 (b) and (c), originally clustered from Graph 3 in Figure 6.8(a), vary in colour and size to highlight the relationships between various abstract levels of the layouts.
CHAPTER 6. SMK: CLUSTERING FOR LAYOUT

(a) $l = 0$

(b) $l = 1$ (red), $2$ (green) and $3$, with inside labels
In our experiments, the parameter is $\tau = 0$ in the algorithm in Figure 5.1. The increase of the threshold $\tau$, which is related to the standard derivation of node degrees, makes possible the reduction of the number of clusters.

A realistic example of hierarchically abstracting graphs is shown in Figure 6.10, with the original part of Java 1.4 class diagrams in Figure 6.10 (a) and its clustered graphs in Figures 6.10 (b)-(e). The underlying purpose of this visualization is to focus mainly on the exploration of the relationships between Java classes, rather than the representation of their inherent hierarchical structures, which may be suitably displayed by a traditional tree drawing. In order to better reveal the relationships between different abstract level views, slightly varied sizes and colours of the nodes are employed to visually encode them. In particular, the graphs, from the most abstract level view to detailed views, consist basically of the following nodes and edges: the biggest and red, the second biggest and green, the second smallest and blue, and the smallest and white nodes. The
readability of the layout can be significantly improved in the presence of only one abstract level view such as the biggest and red nodes. Moreover, in our prototype, users can interact with the layouts through the operations of MMD presented in the next chapter.

(a) Layout of Java 1.4 class diagrams by the Spring algorithm
(b) Layout of all the abstract level without the labels of Java class names ($l=0$)

(c) Layout of the most abstract level with Java class names ($l = 0$)
(d) Layout of the second abstract level with Java class names ($l = 1$)
We have so far presented two approaches, KNN and SMK, in this and the last chapters respectively, to identify the highly linked subgraphs in a graph. The two approaches have a common purpose of reducing the number of nodes and edges by clustering a graph. Also, both approaches start with the identification of Seed Nodes, which are the special nodes with relatively high degrees in a graph. Every seed node is regarded as the first member of a cluster. The number of clusters is equal to that of the Seed Node set. However, the differences between KNN and SMK are as follows.

In essence, KNN is a gradual approach using an affinity function of measure of the links of nodes. In contrast, SMK encodes linkage patterns of a graph into a similarity
matrix, and then processes this matrix to group corresponding highly connected subgraphs as clusters.

The clustered graph by KNN may have singleton clusters, because the process of gradually clustering may stop when the algorithm meets nodes which have equal affinities to existing clusters. On the contrary, such a case cannot happen in SMK, since SMK uses an object function to control when the clustering should stop during the clustering.

As far as the time complexity is concerned, KNN has the advantage over SMK.

6.8 Related Work

Although there are numerous algorithms for cluster analysis in the literature, we briefly review those that are closely related to the structure of a graph, and compare them with our approaches.

Matula [Matula 1970, 1969, 1987, 1972] used a high connectivity in similarity graphs to cluster analysis, which is based on a cohesiveness function. This function defines every node and edge of a graph to be the maximum edge-connectivity of any subgraph containing that element. The $k$-connected subgraphs of the graph are obtained by deleting all elements with cohesiveness less than $k$ in the graph, where $k$ is a constant value. It is, however, hard to determine the values of connectivity in real clustering applications with this approach.

There is some recent work related to the clustering of a graph. The HCS algorithms [Erez, Hartuv, Shamir 2000] use a similarity graph as the input data, partitioning recursively a current set of elements into two subsets. Highly connected subgraphs are then identified as kernels that are considered as clusters, if the number of their edges exceeds half that of their corresponding nodes. Unfortunately the result of the clustering is not uniquely determined. Using the same basic scheme as HCS to form kernels, the CLICK algorithm [Sharan and Shamir 2000] builds on a statistical model, including the following processing: singleton adoption, a recursive clustering process on the set of
remaining singletons, and an iterative merging step. The CAST [Ben-Dor 1999] starts with a single element and uses a single parameter $t$. Elements are added or removed from the cluster if their affinity is larger or lower than $t$, until the process stabilizes.

The features of our approach differ from previous work in the following aspects:

1. Our approach does not require the initial layout of a graph and are suitable for clustering any types of graphs, including tree graphs, since it is based purely on the structure of a graph. Actually, the approach presented here is a novel method of graph partitioning so that it can be applicable to graph partition as well.

2. Without the need to be specified in advance, the number of clusters is automatically detected on the basis of the distribution of node degrees of a given graph--which differs from the $k$-means algorithm.

3. The number of clusters can be easily adjusted by changing the thresholds.

4. More importantly, a metric of measurement of node similarity within a graph is provided. This may result in many potential application areas, such as pattern recognition, and the clustering of Web pages.

6.9 Summary

Graph clustering has applications in many areas such as hypermedia systems, Web communities and graph drawings. This chapter has presented a new approach to clustering a graph. This approach constructs the node similarity matrix of a graph based on a novel metric of node similarity, and then applies the $K$-means algorithm to such a matrix to obtain a hierarchical abstraction of the graph. A heuristic method is also proposed to overcome the inherent drawbacks of the $k$-means algorithm. The proposed approach has demonstrated good results in our experiments, and an example of visualizing part of Java 1.4 class diagrams is shown as well. In order to reduce visual
complexity, we will describe in the next chapter a multilevel multi-window approach to hierarchical layouts of clustered graphs for different abstract level views.
Chapter 7

MMD: Layout of Clustered Graphs

One of the major concerns in graph drawing is the automatic layout of clustered graphs within a screen. There are not very many algorithms available to tackle this problem. In this chapter, we provide a method for the layout of clustered graphs with illustrated examples.

This chapter deals mainly with the issues in layout and view in the proposed framework in Figure 3.3. The main content of this chapter has been published in [Huang and Lai 2002].

7.1 Related Work

From the framework proposed in Chapter 3, we now move on to the layout of graphs after the graphs have undergone the processes of filtering and clustering. In this chapter, our contribution lies in presenting a new method for multilevel multi-window layouts of hierarchically clustered graphs.

As noted in Definition 6.4 in the last chapter, P. Eades et al. [Eades and Feng 1997] defined a clustered graph as follows: A hierarchically clustered graph $C = (G, T) = (\text{graph } G + \text{ tree } T)$ consists of a graph $G$ and a rooted tree $T$ whose leaves are exactly the nodes of $G$. $T$ is called the inclusion tree, and represents a recursive inclusion relation called a clustering of $G$. Internal nodes of $T$ are called abstract nodes or clusters. In other words, the tree structure represents the level of abstraction. For the layout of such clustered graphs, Eades and Feng [Eades and Feng 1997] also proposed a multi-level drawing of a clustered graph $C=(G,T)$, which includes:

- A sequence of the $x$-$y$ plane to represent a drawing of each view from the leaf level (level 0) to the root level of $T$, where the view of abstract level $l$ is drawn on the plane $z = l$.
- A three dimensional drawing of $T$, with each clustered node of $T$ of abstract level $l$ drawn as a point on the plane $z = l$. 

CHAPTER 7. MMD: LAYOUT OF CLUSTERED GRAPHS

An example of multilevel drawing provided by Eades and Feng is shown in Figure 7.1.

The obvious advantage of this approach is that a number of all clustered subgraphs can be simultaneously drawn on one screen. It is, however, impossible to entirely display a large graph using such an approach. Our approach overcomes this drawback by utilizing multilevel and multi-window displays.

7.2 Multilevel Multi-Window Drawings

After a clustered graph has been obtained by applying the algorithms proposed in Chapters 5 and 6, the further issue is how to display it in a way that maximizes readability.

Our approach called MMD gradually displays the multilevel of abstract views of a clustered graph using multi-windows. The basic characteristics of MMD are as follows:

- A sequence of windows in the x-y plane to display a drawing of each view from the leaf level (level 0) to the root level of a tree $T$, where the view of abstract level $l$ is drawn on the $l$-th window. The $z$-level layer of a window corresponds to the

Figure 7. 1: A multilevel drawing (From Eades and Feng [1997])
abstract level of a clustered graph. The bigger the value of $z$, the more abstract the clustered graph.

- The windows can be interactively opened or closed by users.
- There are two display modes: one in which users navigate to a lower (or higher) abstract level view by opening (or closing) an abstract node to display its cluster member nodes; another in which users are allowed to expand or collapse all abstract nodes and edges in a specified abstract level.

With the above descriptions, the MMD can be defined as:

**Definition 7.1 MMD**: Given a graph $G=(V,E)$, MMD refers to a sequence of derived graphs, which can be recursively generated by using a clustering algorithm:

1. $G: G^0=(V^0,E^0), G^1=(V^1,E^1), \ldots, G^l=(V^l,E^l)$
   where $G^{l+1} = C(G^l)$, and $l$ is an integer

and another sequence of display windows:

2. $W^0, W^1, \ldots, W^k$ where $k$ is an integer

as well as their relationships:

3. $G^l \rightarrow W^k$

   $|V^l| \rightarrow sizeOf(W^k)$ and $k \geq l$

In other words, the graph $G^{l+1}$ is derived from the lower abstract level of the graph $G^l$ through clustering. All composition node members of a non-singleton cluster will be replaced with an abstract node in $V^l$ and the edges with an abstract edge in $E^l$. Such a resulting coarse graph becomes the initial graph for the $l+1$th abstract level drawing. Obviously, we have $|V^{l+1}| \leq |V^l|$. The $i$-th abstract level of the graph $G^i$ ($i=0,\ldots,l$) will be separately drawn in different windows, the sizes of which are dynamically determined by both the number of nodes in the graph and users’ interaction.

**Definition 7.2 Operations of MMD**: The typical operations on MMD are:

- *Open*(v): where $v \in V^l$: Open an abstract node $v$ to display all its cluster member nodes and associated edges.
- *Close*(v): where $v \in V^l$: Contract all current display nodes and edges in a cluster into an abstract node and edge.
• **ExpandAll**($G^l$): Layout all nodes and edges including abstract nodes and edges in $G^l$ in a current display window.

• **CloseAll**($G^l$): Close all currently opening abstract nodes and edges.

• **Shrink**($G^l$): Shrink currently opening windows in order to free up more space for the focus window.

In the following, we illustrate some of the above defined operations on MMD in the context of users’ exploration of a clustered Web graph.

Suppose that Figure 7.2 is part of a clustered Web graph, where yellow nodes represent abstract nodes. Two possible operations are associated with an abstract node. **Open**: users can navigate to next or upper abstract level of hierarchical details by clicking on a node. **Close**: users can close opening windows or shrink a group of children nodes into an abstract parent node, to free up display space for other nodes. The resulting layout is shown in the “Wei Lai” node in Figures 7.2 (a) and (b). Also, windows have two operations. **Expand**: within a window, users can continue to open the abstract nodes (see Figure 7.2 (c)), or to explore other invisible nodes that are the neighborhood nodes of the current focus node. **Shrink**: with two or more windows being opened at the same time, non-focus windows may be resized automatically while exploring the nodes within the focus window. This leads to free more display space for the focus window.

Two implemented operations of MMD are shown in Figures 7.3, 7.4 and 7.5, with the **Open** operation shown in Figure 7.4 and the **Close** operation shown in Figure 7.5. In particular, Figure 7.3 illustrates an overall picture of MMD. Figures 7.4 (a) and (b) shows the result of opening the abstract node labeled “current”, while Figure 7.5 (d) is obtained as a result of sequentially closing the two nodes labeled “program” in Figure 7.5 (a) and “current” in Figure 7.5(c).
CHAPTER 7. MMD: LAYOUT OF CLUSTERED GRAPHS

(a) Part of a Web graph

(b) Open operation

(c) Multilevel windows

Figure 7.2: Multiple Level Layouts
CHAPTER 7. MMD: LAYOUT OF CLUSTERED GRAPHS

Figure 7.3: Implementation of MMD
Figure 7.4: The Open operation of MMD
CHAPTER 7. MMD: LAYOUT OF CLUSTERED GRAPHS

(b) Apply Spring Algorithm
Apply Force-Transfer Algorithm
Close

(c)
7.3 Algorithms

Due to its specific characteristics, the layout of a clustered graph should follow the following drawing rules, except for the common drawing aesthetics depicted in Chapter 2:

- **R1**: Incremental display
- **R2**: Hierarchical display
- **R3**: The layout of windows should show the directions of users’ exploration
- **R4**: Adjust dynamically the layout and keep the mental map

The following window expand/shrink algorithm is used for satisfying R1, R2 and R3. The Force-Transfer algorithm (Chapter 8) is applied to meet rule R4, and Modified Spring algorithm is employed as the basis of the layout.

### 7.3.1 Window Expansion/Shrink

The size of a display window in MMD is automatically adjusted by a window expansion/shrink algorithm, which is based on users’ interaction. For example, when
clicking nodes near the boundary of a window, the window may be expanded, which leads to shrink other non-focus windows.

Suppose that a current active window is shown in Figure 7.6, with the current focus node $v$, an abstract node containing children nodes.

![Figure 7.6: Expansion of the focus window](image)

The polar coordinate of node $v$ is $(r, \theta)$. When expanding a node $v$ by means of $\text{Open}(v)$, all the nodes in the corresponding cluster $C_i$ will be displayed in the area around node $v$. The size of the display area should be proportional to the number of nodes in $|C_i|$. If there is not enough space available, the current window will be expanded. The degree of expanded length and width of the window can be calculated by:

$$
\Delta l_r = \begin{cases} 
  d-(l/2-r\cos\theta) & \text{if } d > w/2-r\sin\theta \text{ and } d-(l/2-r\cos\theta) \leq L \\
  0 & \text{otherwise}
\end{cases}
$$

$$
\Delta l_L = \begin{cases} 
  d-(l/2+r\cos\theta) & \text{if } d > l/2+r\cos\theta \text{ and } d-(l/2-r\cos\theta) \leq L \\
  0 & \text{otherwise}
\end{cases}
$$

where:

- $d$: The radius of a circle centered at node $v$, i.e., $d = k\sqrt{|C_i|}$ ( $k$ is a constant)
- $\Delta l_r$: The expanded distance of the window to the right
- $\Delta l_L$: The expanded distance of the window to the left
- $L$: The maximum length of the screen display window
CHAPTER 7. MMD: LAYOUT OF CLUSTERED GRAPHS

W: The maximum width of the screen display window

Similarly, we have the following formula to compute expanded distances of windows in up and down directions:

$$
\Delta w_v = \begin{cases} 
  d - (w/2 - r \sin \theta) & \text{if } d > w/2 - r \sin \theta \text{ and } d - (w/2 - r \sin \theta) \leq W \\
  0 & \text{otherwise}
\end{cases}
$$

$$
\Delta w_d = \begin{cases} 
  d - (w/2 + r \sin \theta) & \text{if } d > w/2 + r \sin \theta \text{ and } d - (w/2 + r \sin \theta) \leq W \\
  0 & \text{otherwise}
\end{cases}
$$

Obviously, the positions of a focus node v and the number of nodes in its cluster \(|C_i|\) determine the expansion directions of a window. Such directions simply fall into the following categories: right, upper-right, upper, upper-left, left, down-left, down, and down-right. The non-focus windows will shrink proportionally until they contract into nodes.

7.3.2 The Modified Spring Algorithm

In order to show the direction of exploration, the Modified Spring Algorithm [Huang and Eades 1998] adds extra forces among the neighbors of focus nodes. There forces separate the neighbors so that users can easily identify the change caused by shifting the focus node. The applied force on node v is:

$$
f(v) = \sum_{u \in N(v)} f_{uv} + \sum_{u \in V_0} g_{uv} + \sum_{u \in Q} h_{uv}
$$

Where \(f_{uv}\) is the spring force on node v produced by both node u and node v, \(g_{uv}\) and \(h_{uv}\) are the Newtonian gravitational repulsion forces on node v by other node u.

We find all the nodes in a current active window and then apply this algorithm to layout them.
7.3.3 Dynamic Adjustment and Remove Overlapping Nodes or Windows

The Modified Spring Algorithm is suitable for the initial layout of a clustered graph. However, there are cases in which a layout is required to be adjusted, especially in the course of users’ exploration and navigation. These include the following scenarios:

- **Open** operation: The unfolding of an abstract node may introduce overlaps with other existing nodes in the same display window.
- **ExpandAll** operation: A newly created display window may lead to overlapping with other nodes or windows.

In these cases, the original graphs or windows should be repositioned to satisfy two criteria:

- The nodes and/or windows in a new layout should no longer be overlapped
- The adjustment of an original layout should be made as little as possible

In the following chapter, a new algorithm [Huang and Lai 2003] is provided to satisfy the above requirements.

7.4 Summary

To reduce visual complexity, we have described a multilevel multi-window approach to the hierarchical layout of clustered graphs in different abstract level views. Improving the 3D drawing approach of P. Eades et al., our approach incorporates methods for hierarchically organizing nodes, a dynamic adjustment algorithm for expansion/shrink of multilevel display windows, and a Force-Transfer algorithm\(^1\).

By comparison with other approaches, our proposed approach has following salient features of both incrementally and interactively drawing graphs:

- Improve the readability of a layout by incrementally displaying clustered graphs as well as by employing multilevel, multiple window displays.

\(^1\) This algorithm will be presented in the next chapter.
CHAPTER 7. MMD: LAYOUT OF CLUSTERED GRAPHS

• Show the explorative directions of huge graphs by means of the layouts of both nodes and windows.
• Maintain the mental map due to a compact adjustment.

A new algorithm used in MMD for layout adjustments will be presented in the next chapter.
Chapter 8

Force-Transfer: Removing Overlapping Nodes

When applying layout algorithms to attributed graphs defined in Chapter 3, the problem of overlapping nodes may occur, particularly for nodes with inside labels, namely simple attributed nodes or labeled nodes. While users are interacting with a layout, the changes to the layout should be minimized to preserve the users’ mental map. Although there are several algorithms to resolve this problem, the new approach proposed in this chapter can achieve better results than these algorithms with respect to compact adjustments and reasonable computational time. A detailed description of the approach called Force-Transfer Algorithm (FTA) will be provided and illustrated with examples.

This chapter deals mainly with the issues in layout and view in the proposed framework in Figure 3.3. The main content of this chapter has been published in [Huang and Lai 2003].

8.1 Introduction

The nodes in a graph represent objects or entities, which have distinct labels as their identifiers. These labels in a drawing can be presented in the form of texts, digits, or even images. They should be drawn not as abstract points with almost no size that most drawing algorithms assume, but rather as rectangles large enough to display the labels. For example, UML diagrams used in CASE tools are labeled graphs. Those graphs with labels are referred to as practical graphs [Lai and Eades 2002]. When applying a traditional algorithm for graph layout to such a labeled graph, a node overlapping problem may occur, destroying the layout aesthetics—the main purpose of the layout algorithm. The need for removing overlapping nodes thus arises. Furthermore, in a dynamic environment where a graph can often be modified such as enlarge/shrink sub-graph and add/delete nodes, the layout of the graph should be adjusted to cater for these changes. The techniques for removing overlapping nodes can also be directly applied to position overlapping windows in multi-window
applications, and to layout information for small devices such as mobile phones and PDAs. When eliminating overlapping nodes, it is desirable that adjustment of the original layout should be kept to a minimum. Preserving the mental map [Lyons 1998; Misue et al. 1995] of an original layout, a generic approach for practical graph layouts has been proposed: apply an existing graph drawing algorithm to a practical graph first, and then use a post-process to remove overlapping nodes and edge-node intersections [Lai and Eades 2002].

The remainder of this chapter is organized as follows: the following section reviews related work. Section 8.3 describes FTA in detail, followed by the comparison FTA with FSA (Force-Scan Algorithm) in Section 8.4. The properties of FTA and their proofs are presented in Section 8.5. In Section 8.6, we provide empirical evaluation by applying both FTA and FSA to a set of arbitrary graph layouts. Some potential applications are discussed in Section 8.7. Finally, we summarize and conclude this chapter in Section 8.8.

8.2 Related Work

There are essentially three kinds of approaches to solving the node-overlapping problem: Uniform Scaling [Lai 1993; Eades 1992], Constrained Optimization [Marriott 1998; Marriott 2003] and Force-Based Algorithms [Eades 1992; Lyons 1998]. We review these approaches in the following paragraph.

Preserving the original structure of a graph, a straightforward approach called Uniform Scaling avoids node overlaps by uniformly scaling the overlapping layout. The nodes may, however, be expanded unnecessarily and the adjusted layout thus tends to be too large.

The Constrained Optimization approach views removing overlapping nodes as a problem of constrained quadratic optimization. The object function comprises fundamentally a quadratic expression about differences between the initial and adjusted coordinates of the nodes within a graph. This approach provides an optimal solution to such an object function, subject to a collection of constraints that ensure no
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

node overlapping [Marriott 1998; Marriott 2003]. As an example, a layout procedure of such an approach can be divided into two phases. In the first phase, an aesthetic cost function suggested by Kamada [Kamada 1989] is used to quickly obtain a certain local minimum, and then check whether node overlapping occurs or not. If not, the layout terminates; otherwise, the second phase starts to shift overlapping nodes slightly along different directions in order to move every node to a different point, and then to use the cost function to finalize the layout. There are also several variations of this approach [Marriott 1998]. In general, this kind of approaches can “give better layout than the force scan algorithm, although they are slower” [Marriott 2003].

The Force-Based Algorithm for addressing the overlapping node problem mainly includes Cluster Busting in an anchored graph drawing [Lyons 1998] and the Force-Scan Algorithm [Lai 1993; Misue 1995; Eades 1992] (FSA). The procedure of Cluster Busting is iterative because the nodes in a graph are iteratively relocated in accordance with measurable criteria. Those criteria are introduced to improve the distribution of nodes (cluster busting) and to minimize the difference between the original layout and adjusted layout (anchored graph drawing). Also, those algorithms have to run several iterations to achieve a better-adjusted layout.

FSA, a classical algorithm, is widely used in applications, on which our approach is based. The main idea of FSA is to apply a ‘force’ \( f_{vu} \) to one of two overlapping nodes \( v \) and \( u \) so that \( f_{vu} \) pushes node \( u \) away from node \( v \). The forces for all the nodes are applied in sequence within two scans: one left-to-right horizontal scan, and another top-to-bottom vertical scan. In other words, FSA moves the nodes in both horizontal and vertical directions to avoid overlaps. Preserving the mental map of the original graph, this approach produces a compact layout compared with Uniform Scaling.

A variant of FSA allows an additional pull force between two nodes, making the graph layout as compact as possible. This approach with push and pull forces, however, needs to run a number of times of the original FSA until the forces among nodes gradually become zero [Eades and Lai 1992], so the speed of this approach is relatively slow.

More recently, David Harel and Yehuda Koren [Harel and Koren 2002] proposed a modified spring method to prevent overlapping nodes at the beginning of a layout,
considering the node size in the original spring embedded algorithm. Nevertheless, this approach cannot be efficiently applied to a dynamic graph layout, because it needs running from scratch whenever a change is made in the graph. This results in too much cost of time and computation.

Other related work includes the SHriMp Algorithm [Müller 1995], where the nodes in a graph uniformly give up their screen space to allow a node of interest to grow. These nodes are appropriately scaled to fit inside the screen. This approach is thus a distortion technique.

This chapter presents a new approach called Force-Transfer Algorithm (FTA) to removing overlapping nodes in graph layouts. This approach which has the same time complexity as FSA can produce a more compact layout than FSA. In other words, FTA can not only remove all overlapping nodes in a graph, but also make a compact adjustment of the layout.

Using a heuristic method, FTA approximates a global optimal adjustment with local minimal movements. It addresses such three questions: how great a force should be, in which way an applied force is transferred to other nodes, and where to start with the first force. The answers that follow to these questions make a distinction between FTA and FSA. First of all, a minimum force in FTA is applied to separate two nodes in an orthogonal direction so that it can achieve the local optimisation of an adjustment. Second, the force is restrictedly transferred to a dynamic subgraph that is a group of nodes overlapping each other. It is possible that a node does not overlap with any node in the subgraph, but does with at least one of them after an adjustment. The nodes in the subgraph are therefore updated iteratively during the scan. The process of the adjustment will continue until such subgraphs including overlapping nodes no longer exist in the final layout. It should be noted that a node with the initial applied force, called the Seed Node in FTA, can be any node in the graph. Our prototype system allows users to interactively select a Seed Node for a graph layout. This can make the whole graph adjust as little as possible while removing overlapping nodes.
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

8.3 The Force-Transfer Algorithm

We assume that there is an overlapping graph \( G = (V, E) \), where \( V \) is the set of nodes, and \( E \subseteq V \times V \) is the set of edges. Let \( (x_v^0, y_v^0) \) denote the centre of a node \( v \) with a rectangular bounding box of width \( w_v \) and height \( h_v \), and denote \( (x_v^1, y_v^1) \), \( (x_v^2, y_v^2) \), and \( (x_v^3, y_v^3) \) as its four corner positions, respectively (see node \( v \) in Figure 8.1).

The nodes in \( G \) are sorted by \( x_1^i \) (the coordinates of upper left corner \( (x_1^i, y_1^i) \) of node \( v_i \)) and then by \( y_1^i \) in horizontal and vertical directions, where \( i = 1, 2, \ldots, |V| \).

8.3.1 Definitions and Problem Statement

With the above preliminaries, we can provide definitions and formally describe the overlapping problem in this section.

**Definition 8.1 Neighbor Node:** Two nodes \( v \) and \( u \in V \), with the following expression being satisfied:

\[
|x_v^0 - x_u^0| \leq (w_v + w_u)/2 + d \quad \text{or} \quad |y_v^0 - y_u^0| \leq (h_v + h_u)/2 + d
\]

are called the Neighbor Node denoted by \( n(u, v) \), where \( d \) is a minimum gap between two nodes \( v \) and \( u \). To simplify notation, the gap \( d \) is omitted in the following formulae.

**Definition 8.2 Neighbor Nodes:** A set of the nodes that overlap with node \( v \), Formally, \( NS(v) = \{ u \mid n(v, u), u \in V \} \) where \( NS(v) \) denotes the Neighbor Nodes of node \( v \).

![Figure 8.1: Neighbor Nodes](image)

Considering the example in Figure 8.1, we have \( NS(v) = \{ u_1, u_2, u_3 \} \). In order to describe our algorithm later, the definitions of *Left, Right, Up and Down Neighbor Nodes* of a node \( v \) are given by:

\[
\begin{align*}
LNS(v) & = \{ u \mid x_u^1 < x_v^1, u \in NS(v) \} , \\
RNS(v) & = \{ u \mid x_u^1 \geq x_v^1, u \in NS(v) \} \\
UNS(v) & = \{ u \mid y_u^1 < y_v^1, u \in NS(v) \} , \\
DNS(v) & = \{ u \mid y_u^1 \geq y_v^1, u \in NS(v) \}
\end{align*}
\]
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

Obviously, we have \( LNS(v) = \{u_1\} \), \( RNS(v) = \{u_2, u_3\} \), \( UNS(v) = \emptyset \), and \( DNS(v) = \{u_1, u_2, u_3\} \) in Figure 8.1.

**Definition 8.3** *Transfer Neighbor Nodes*: Denoted by \( TNS(v) \), a set of nodes near node \( v \) including all nodes around node \( v \) in which every node at least overlaps with one other node. The identification of a \( TNS(v) \) is performed as follows: find *Neighbor Nodes* of a node \( v \), i.e. \( NS(v) \), and then all the nodes in \( NS(v) \) in turn find their *Neighbor Nodes* and so on, until a particular node with no *Neighbor Nodes* is met. Formally, the \( TNS(v) \) can be recursively defined in this way:

\[
i = 1: TNS(v)_i = \bigcup_{u_j \in NS(v)_i} NS(u_j) \cup NS(v) - \{v\} \\
i > 1: TNS(v)_{i+1} = \bigcup_{u_j \in TNS(v)_i} NS(u_j) \cup TNS(v)_i - \{v\}
\]

If the equation \( TNS(v)_i = TNS(v)_{i-1} \) is always satisfied, then

\( TNS(v) = TNS(v)_1 \)

where \( i = 1, 2, \ldots, |V| - 1 \).

Again referring to Figure 8.1, we have:
\[
TNS(v)_1 = NS(u_1) \cup NS(u_2) \cup NS(u_3) \cup \{u_1, u_2, u_3\} - \{v\}
\]

and
\[
TNS(v)_2 = NS(u_1) \cup NS(u_2) \cup NS(u_3) \cup NS(u_4)
\]

\[
\cup \{u_1, u_2, u_3, u_4\} - \{v\}
\]

Eventually, the *Transfer Neighbor Nodes* of node \( v \) is obtained: \( TNS(v) = \{u_1, u_2, u_3, u_4, u_5\} \).

Similarly, we can define *Left, Right, Up and Down Transfer Neighbor Nodes*. For instance, we have \( TLNS(v) = \{u_1\} \), \( TRNS(v) = \{u_2, u_3, u_4, u_5\} \), \( TUNS(v) = \emptyset \), and \( TDNS(v) = \{u_1, u_2, u_3, u_4, u_5\} \) in Figure 8.1.

It is clear that the following corollary derived from the above definitions is true:

**Corollary** (1) \( TNS(v) \supseteq NS(v) \); (2) \( TNS(v) = TRNS(v) \cup TLNS(v) \); (3) \( TNS(v) = TUNS(v) \cup TDNS(v) \); (4) \( TRNS(v) \cap TLNS(v) = \emptyset \); (5) \( TUNS(v) \cap TDNS(v) = \emptyset \).

The following theorem will be used in order to significantly reduce the computational complexity of the proposed algorithm.
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

Theorem 8.1 Given a graph layout with overlapping nodes and \( i \neq j \), the following holds true.

1. If \( TNS(v_i) \cap NS(v_j) \neq \emptyset \), then \( TNS(v_i) \supset NS(v_j) \)

Proof. (1) \( TNS(v_i) \cap NS(v_j) \neq \emptyset \) holds, which implies at least one node \( u \) in both \( TNS(v_i) \) and \( NS(v_j) \). This node \( u \) acts as a bridge node to form a Neighbor Node with one node in \( TNS(v_i) \) and \( NS(v_j) \) respectively. This indicates that these three nodes are Neighbor nodes. According to Definitions 2 and 3, the claim is immediate.

(2) Without loss of generality, suppose that we have \( TNS(v_i) \subset TNS(v_j) \), then we can rewrite it as \( TNS(v_i)-TNS(v_i)=TNS(v_i) \), and \( TNS(v_i) \cap TNS(v_k) = \emptyset \). It follows that there must be two nodes \( u \in TNS(v_i) \) and \( w \in TNS(v_k) \) such that \( u \) and \( w \) is a Neighbor Node namely \( N(u,w) \), otherwise \( TNS(v_k) \) cannot be included in \( TNS(v_i) \). On the other hand, since \( N(u,w) \) is true, \( TNS(v_k) \) should be included in \( TNS(v_i) \) according to the definition of Transfer Neighbor Nodes. That is, \( TNS(v_i) \cap TNS(v_k) \neq \emptyset \) holds. This results in a contradiction. Therefore the claim is correct.

Definition 8.4 Cost Function: A function used to measure the degree of an adjustment, the value of which equals the sum of the moved distances over all the nodes in \( V \) after a layout adjustment:

\[
f_{\text{cost}}(v_i) = \sum_{v \in V} \Delta x_v + \Delta y_v
\]

where \( \Delta x_v \) and \( \Delta y_v \) denote the orthogonally moved distances of a node \( v \). This function is clearly related to the Seed Node, which is defined as follows.

Definition 8.5 Seed Node\(^1\): A node \( v_s \in V \), or a dummy node in a graph to which the first force in an adjustment is applied. As a dummy node, i.e. a point without size, it is actually a reference point used as the benchmark of a scan.

Now, the problem of removing overlapping nodes can be formalized.

Definition 8.6 Removing Node Overlap Problem: Given a graph \( G \) with an overlapping node layout, an adjustment to removing node overlap should satisfy the following conditions:

1. \( NS(v) = \emptyset \), for \( \forall v \in V \)
2. \( \min_{v \in V} \{f_{\text{cost}}(v_s)\} \)

Note that this definition is different from Definition 5.6 in Chapter 5.
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

More strictly, a mental map [Lai 1993; Misue et al. 1995] of the original layout should be preserved in some applications.

With the above definitions, we can describe our new approach. According to Definition 8.1, two nodes $v$ and $u \in V$ are separated such that the following expression should hold:

$$\left| x_v^0 - x_u^0 \right| \geq \left( w_v + w_u \right) / 2 \quad \text{or} \quad \left| y_v^0 - y_u^0 \right| \geq \left( h_v + h_u \right) / 2$$

Consider a simple case of two overlapping nodes in Figure 8.2(a). With node $v$ remaining unchanged, a force on node $u$ can be randomly applied from anywhere to eliminate the overlaps. The orthogonal projection of this force, however, must push $u$ a distance at least either $\Delta x_{uv}$ to the right/left or $\Delta y_{uv}$ down/up. FTA initially calculates the orthogonally overlapping distances between two nodes $v$ and $u$:

$$\Delta x_{uv} = \min_{n(v,u)} \{ |x_v^0 - x_u^0| \}$$

$$\Delta y_{uv} = \min_{n(v,u)} \{ |y_v^0 - y_u^0| \}$$

and then chooses the minimum magnitude of $\Delta x_{uv}$ and $\Delta y_{uv}$ as a “force” to work on node $u$ in order to push node $u$ away from node $v$:

$$|f_{uv}| = \min_{n(v,u)} \{ |\Delta x_{uv}, \Delta y_{uv}| \}$$

Considering the direction of the force, we can rewrite the above equation as:

$$f_{uv} = \begin{cases} 
\Delta x_{uv} \hat{i} & \text{if } \Delta x_{uv} \leq \Delta y_{uv} \land x_v^0 \geq x_u^0 \land x_v^0 \geq x_u^0 \\
-\Delta x_{uv} \hat{i} & \text{if } \Delta x_{uv} \leq \Delta y_{uv} \land x_v^0 < x_u^0 \land x_v^0 < x_u^0 \\
\Delta y_{uv} \hat{j} & \text{if } \Delta x_{uv} > \Delta y_{uv} \land y_v^0 \geq y_u^0 \land y_v^0 \geq y_u^0 \\
-\Delta y_{uv} \hat{j} & \text{if } \Delta x_{uv} > \Delta y_{uv} \land y_v^0 < y_u^0 \land y_v^0 < y_u^0 \\
0 & \text{otherwise}
\end{cases}$$

where $\hat{i}$ or $\hat{j}$ is a unit vector in the $x$ or $y$ direction.

In the example of Figure 8.2 (b), the $x$ and $y$ projections of the applied force are $|f_x^*| = dx_{uv}$, and $|f_y^*| = 0$, due to different overlapping distances in the two directions ($\Delta x_{uv} < \Delta y_{uv}$), whilst in Figure 2 (d), $\Delta x_{uv} > \Delta y_{uv}$ leads to $|f_x^*| = dy_{uv}$ and $|f_y^*| = 0$. We correspondingly apply the above forces $f_{ux}^*$ and $f_{uy}^*$ to
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

the respective graphs shown in Figures 8.2 (b) and (d), and arrive at the resulting layouts given in Figures 8.2 (c) and (e), respectively.

Figure 8.2: Overlap calculation

Sequentially computing the orthogonal overlap distance of every pair of neighbor nodes in a graph, the scan starts from the Seed Node to those nodes located at to the right, left, above and below.

8.3.2 Force-Transfer Model

Our approach starts with the choice of a Seed Node, and then performs four scans in the four directions. The Force-Transfer model is applied to the process of such each scan. In this section, we use the right scan as an example to explain it.

We define a cluster sub-graph denoted by \( C(v) = TNS \ (v) \cup \{ v \} \) as a cluster of overlapping nodes (see Figure 8.3). To get the boundary of such a cluster sub-graph, we use a rectangle to include the nodes in this sub-graph. This rectangle can alternatively be regarded as one new cluster node (i.e. a dummy node), denoted by \( v_C \).

The dash-line box in Figure 8.3 indicates a dummy node, with its four corner positions \((x^1_{v_c}, y^1_{v_c}), (x^2_{v_c}, y^1_{v_c}), (x^2_{v_c}, y^2_{v_c})\) and \((x^1_{v_c}, y^2_{v_c})\), where \(x^1_{v_c}, y^1_{v_c}, x^2_{v_c}\) and \(y^2_{v_c}\) can be calculated:

\[
x^1_{v_c} = \min_{i \in C(v)} \{x^i_{v_c}\}, \quad y^1_{v_c} = \min_{i \in C(v)} \{y^i_{v_c}\}, \quad x^2_{v_c} = \max_{i \in C(v)} \{x^i_{v_c}\}, \quad y^2_{v_c} = \max_{i \in C(v)} \{y^i_{v_c}\}
\]

The graph \( G \) can be accordingly treated as a union of \( C(v), C(u), \ldots \), etc. (Figure 8.3). Suppose that the Seed Node \( v_s \) is included in \( C(v) \). The scan begins with this Seed Node, and the scan direction is from left-to-right.
Figure 8.3: Cluster nodes and force transfer

During this right scan, the first applied forces are transferred to the nodes in \( \text{TRNS}(v_i) \). That is, the force \( f_{ij}^* \) between nodes \( v_i \) and \( v_j \) in \( \text{TRNS}(v_i) \) will also be transferred to the nodes in \( \text{RTNS}(v_j) \). As a result, the total force on node \( v_t \in C(v) \) is as follows:

\[
F_i^* = \sum_{j=1}^{s} f_{ij}\]

where \( s < i \leq |C(v)| + k - 1 \). \( k \) is the first subscript of all ordered nodes in \( C(v) \) and \( f_{j,j+i}^* \) is calculated by formula (8.4).

Since the overlapping nodes are repositioned in the course of the scan, the width of a cluster node \( v_C \) may be expanded in the right direction. Assume that after scanning node \( v_i \), the width of \( v_C \) has been expanded by \( \Delta w_{v_C} \), then we have:

\[
\Delta w_{v_C} = |F_i^*| = |\sum_{j=1}^{s} f_{ij}|\]

There exists an integer \( k_0 \) (\( s \leq k_0 \leq i - 1 \)) such that the following expression holds true:

\[
\Delta w_{v_C} \geq d^* = \text{dist}(v_C, u_C) - (w_{v_C} + w_{u_C}) / 2 \quad (8.5)
\]

where \( \text{dist}(v_C, u_C) \) is the distance between central positions of cluster nodes \( v_C \) and \( u_C \), with \( w_{v_C} \) and \( w_{u_C} \) as their original widths, respectively. This means that one of cluster nodes \( v_C \) and \( u_C \) becomes a Neighbor Node. Consequently, all applied forces between the nodes in \( \text{TRNS}(v_{i+1}) \), i.e. from node \( v_i \) to node \( v_{E|(v)+k+i} \) in \( C(v) \), will also be applied to all the nodes in \( C(u) \). In other words, a new cluster node is dynamically created, consisting of both the nodes in \( \text{TRNS}(v_{i+1}) \) and those in \( C(u) \).

We conclude that the transfer function that calculates totally applied forces on node \( v_i \) in \( C(v) \) or \( C(u) \), is as follows:

\[
F_i^* = \begin{cases} 
\sum_{j=1}^{s} f_{ij}^* & s \leq i \leq |C(v)| + k - 1 \\
\sum_{j=k+i}^{s} f_{ij}^* & l \leq i \leq |C(u)| + l - 1 \quad \text{if} \quad (2.5) \text{ is satisfied} \\
\sum_{j=k+i}^{s} f_{ij}^* & l \leq i \leq |C(u)| + l - 1 \quad \text{if} \quad (2.5) \text{ is not satisfied}
\end{cases} \quad (8.6)
\]
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

where \( s \leq k_i \leq i - 1 \), and \( k_0 \) is an integer, and \( l \) is the first subscript of all ordered nodes in \( C(u) \).

The similar transfer functions of the left, up and down scan directions can be easily given.

The transfer functions are employed to dynamically find the Transfer Neighbor Nodes in FTA.

8.3.3 Pseudo-Code of the Force-Transfer Algorithm

In the following pseudo-code, assume that given \( G = (V, E, v_s) \), and \( x^1_i \) denote the upper left corner \( x \) coordinates of the nodes, where \( i = 1, 2, \cdots, \left|V\right| \), and \( v_s \) is the Seed Node. The Horizontal Transfer is carried out in two directions, namely from the Seed Node to the right nodes, and from the Seed Node to the left nodes. Here we use the former as an example.

Right Horizontal Transfer:

---

**Input:** A graph \( G \) layout and Seed Node \( s \)

**Output:** An adjusted graph layout where all node overlaps on the right side of the Seed Node are removed if their vertical overlapping distances exceed their horizontal ones

\[
i \leftarrow s
\]
Find\_RNS \((G, s)\)
Find\_TRNS\((G, s)\)

while \( i < \left|V\right| \) { Right Horizontal Scan }

if \( RNS(v_i) \neq \emptyset \) then

\[
\begin{align*}
    f_{x_{ij}}^x &:= x_{ij}^2 - x_{ij}^1 \\
    f_{y_{ij}}^y &:= \text{min}\{y_{ij}^2 - y_{ij}^1, y_{ij}^1 - y_{ij}^2\}
\end{align*}
\]

\[
\delta \leftarrow \text{min}(f_{x_{ij}}^x, f_{y_{ij}}^y)
\]

if \( \delta = f_{x_{ij}}^x \) then

// Right Horizontal Transfer

for each \( v_j \in TRNS(v_i) \)

\[
    x_{v_j}^0 := x_{v_j}^0 + \delta + d
\]

end for

end if

//update TRNS
\[
i \leftarrow i + 1
\]
if $N(v_i, u)$ is true and $u \in V - \text{TRNS}(v_i)$ then
$$\text{TRNS}(v_i) = \text{TRNS}(v_i) + \{u\}$$
end if
end if

// Find the Right Neighbor nodes
Find_RNS(G, s) {
    for $i=s$ to $|V|-1$
        $\text{RNS}(v_i) = \{v_i\}$
        for $j=s+1$ to $|V|$
            if $N(v_i, v_j)$ is true then
                $\text{RNS}(v_i) = \text{RNS}(v_i) + \{v_j\}$
            end if
        end for
    end for
}

// Find the Right Transfer Neighbor Nodes
Find_TRNS(G, s) {
    $i=1$
    do
        for $j=i+1$ to $|V|$
            if $\text{TRNS}(v_i) \cap \text{RNS}(v_j) \neq \emptyset$ then
                //According to Theorem 8.1
                $\text{TRNS}(v_i) = \text{TRNS}(v_i) \cup \text{RNS}(v_j)$
            end if
        end for
        $i=i+1$
    if $v_i \in \text{TNS}(v_{i-1})$ then
        //According to Theorem 8.1
        $\text{TRNS}(v_i) = \text{TRNS}(v_{i-1})$
    else if $\text{RNS}(v_i) \neq \emptyset$ then
        $\text{TRNS}(v_i) = \text{RNS}(v_i)$
    end if
end if
while $i \leq |V| - 1$
}

FTA consists of both the horizontal and vertical scans, which are similar to each other. Overall, the force transfer starts from the Seed Node and proceeds to other nodes in four orthogonal directions. The main FTA algorithm is as follows:

FTA algorithm

\begin{itemize}
  \item \textbf{Input:} A graph G layout
  \item \textbf{Output:} An adjusted layout without overlapping nodes
\end{itemize}
Sort all the nodes according to the values of their $x_i$ coordinates

Find or use the leftmost node as the Seed Node

**Right Horizontal Transfer** $(G, s)$

**Left Horizontal Transfer** $(G, s)$

**Up vertical Transfer** $(G, s)$

**Down vertical Transfer** $(G, s)$

## 8.3.4 Choice of Seed Node

From the previous description, it is clear that the Seed Node plays an important role in our approach. A candidate of the Seed Node should enable minimization of the value of the Cost Function. The problem can be viewed as an optimization problem, that is, to minimize the objective function $f_{\text{cost}}(v_j)$, subject to removing all overlapping nodes in a graph. Many methods may be employed to find the solution such as Simulated Annealing [Harel 1996], Genetic algorithm [Petry 1992] and Tabu Search [Glover 1989]. In order to approximate a global optimization, those algorithms try to escape from getting trapped in local minima by different ways. Applying those approaches to our problem, we may find an optimal point as the centre of the Seed Node. In the case of the point not being inside any node in $V$, it can be treated as a dummy node. Determining this optimal point is, however, a time-consuming process. A heuristic method is therefore employed to approximate the optimum. Our experiments have shown that, in most cases, the choice of the leftmost node as the Seed Node can produce a better layout, except that for symmetric overlapping layouts the centre nodes of the graphs should be used instead. Another choice is to let the user interactively select the Seed Node for a fast graph layout.

## 8.4 Comparison with the Force-Scan Algorithm

While FTA is based on FSA, FTA has the following advantages over FSA:

*Applied forces:* Compared with a force in FSA, FTA uses only a minimum force to separate two overlapping nodes, which is actually equal to the minimum of two orthogonal projections of the corresponding force in FSA. One of the overlapping nodes is thus moved only in the horizontal or vertical direction. In contrast, the node in FSA is relocated in both orthogonal directions because the force has two
projections. The differences between two forces in FTA and FSA applied to the same overlapping node are illustrated in Figure 8.4.

More precisely, the forces used in FSA are calculated as follows [Lai 1993; Misue et al. 1995]:

\[
f_{uv} = \max(0, k_{uv} - d_{uv}) \tau_{uv}
\]

where

- \( k_{uv} \): The desirable distance, i.e. the Euclidean distance between the centers of two separated nodes \( u \) and \( v \) after the adjustment
- \( d_{uv} \): The actual distance, i.e. the Euclidean distance between the centers of two nodes \( u \) and \( v \) before the adjustment
- \( \tau_{uv} \): A unit vector in the direction from the centre of node \( u \) to that of node \( v \), indicating the force \( f_{uv} \) direction

**Transfer of the forces:** In the case of the right scan in FTA, a force is only transferred to the nodes within a dynamic subgraph on the right, i.e. to the nodes belonging to the Transfer Neighbor Nodes of a node from which this applied force starts. Conversely, the force on a node in FSA is unconditionally transferred to all nodes to the right of this node.

**Seed Node:** A Seed Node in FTA can be any node in a graph, even a dummy node, while FSA selects only the leftmost one.

**Calculation of a force:** FTA uses the top-left corner coordinates in place of the centre ones of a node for the force calculation.

For these reasons FTA produces a nicer and more compact layout than does FSA.
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

8.5 Properties of the Force-Transfer Algorithm

In this section, some properties of FTA and their proofs are provided.

**Lemma 8.1** Given a graph layout with overlapping nodes, an applied force in FTA is not greater than the corresponding one in FSA, namely $f_{ij}^{FTA} \leq f_{ij}^{FSA}$.

**Proof.** We give the formulas for calculating an applied force used to separate two overlapping nodes $i$ and $j$ in both algorithms. From (8.2), (8.3) and (8.4), we have:

$$
\delta_v = \left| f_v^* \right| = \begin{cases} 
(w_i + w_j)/2 - |x_v - x_i| & \text{if } \Delta x \leq \Delta y \land N(i, j) \\
0 & \text{otherwise}
\end{cases}
$$

and

$$
\delta_v^* = \left| f_v^* \right| = \begin{cases} 
(h_i + h_j)/2 - |y_v - y_i| & \text{if } \Delta x > \Delta y \land N(i, j) \\
0 & \text{otherwise}
\end{cases}
$$

In FSA, we have [Lai 1993, Misue et al. 1995]:

$$
\delta_v = \begin{cases} 
(w_i + w_j)/2 - |x_v^o - x_i^o| & \text{if } \Delta x \leq \Delta y \land N(i, j) \\
\phi_v^o & \Delta x > \Delta y \land N(i, j) \land y_v^o \neq y_i^o \\
0 & \text{otherwise}
\end{cases}
$$

$$
\delta_v^* = \begin{cases} 
(h_i + h_j)/2 - |y_v^o - y_i^o| & \text{if } \Delta x > \Delta y \land N(i, j) \\
\phi_v^o & \Delta x \leq \Delta y \land N(i, j) \land x_v^o \neq x_i^o \\
0 & \text{otherwise}
\end{cases}
$$

where

$$
\phi_v^o = \left(1 + \frac{h_i + h_j}{2(y_i^o - y_j^o)}\right) |x_i^o - x_j^o|
$$

and

$$
\phi_v^o = \left(1 + \frac{w_i + w_j}{2(x_i^o - x_j^o)}\right) |y_i^o - y_j^o|
$$

Comparing (8.10) with (8.11), it is clear that for removing two overlapping nodes $v_i$ and $v_j$, an applied force $\delta_v^* (\text{or } \delta_v^*)$ in FTA is not greater than $\delta_v^* + \phi_v^o (\text{or } \delta_v^* + \phi_v^o)$ in FSA. Only in the case of two overlapping nodes with the same $x$ or $y$ centre coordinates, the corresponding applied forces in FTA and FSA are equivalent.
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

Lemma 8.2 Given that an applied force is transferable, the sum of this transferred force on all nodes in FTA is not greater than those in FSA.

Proof. Suppose that Seed Node is the leftmost node in $V$, i.e., $v_s = v_1$. In the worst case, it is possible that the applied force between nodes $v_i$ and $v_j$ ($j > i$) in both algorithms will be transferred to all the nodes to the right of node $v_j$, namely its Transfer Right Neighbor Nodes denoted as $TRNS(v_j)$. Consequently, the totally adjusted distances in FTA and FSA are generally given by

$$f(v_1) = \sum_{i=1}^{[V]} (\sum_{j=i+1}^{[V]} \Delta_{x_{ij}} + \sum_{j=i+1}^{[V]} \Delta_{y_{ij}}) \quad (8.12)$$

Where $\Delta_{x_{ij}}$ and $\Delta_{y_{ij}}$ denote the orthogonally moved distances of node $j$ in order to separate it from its overlapping node $i$. The overlapping nodes in FTA are repositioned along either the x or y direction, depending on their overlapping distances $\Delta x_{ij}$ and $\Delta y_{ij}$ . Without lost of generality, Suppose that among the total number of $|TRNS(v_i)|$ nodes within $TRNS(v_i)$, there are $m_i$ pairs of overlapping nodes $v_i$ and $v_j$ having $\Delta x_{ij} \leq \Delta y_{ij}$, where $0 \leq m_i \leq |TRNS(v_i)|$ and $m_i$ is an integer.

In order to remove the first overlapping node $v_j$ of node $v_i$, each node in $TRNS(v_i)$ within FTA moves a distance $\delta_{y}^j$ to the right if we have $\Delta x_{ij} \leq \Delta y_{ij}$. All adjusted distances for relocating $v_j$ thus amount to $\delta_{y}^j |TRNS(v_i)|$. Based on the above assumption, $m_i$ pairs of overlapping nodes in $TRNS(v_i)$ result in the movement distances of $m_i \delta_{y}^j |TRNS(v_i)|$. It follows that the rest of the nodes $|TRNS(v_i)| - m_i$ should be repositioned in the y direction. Therefore, we have

$$f^{FTA}(v_1) = \sum_{i=1}^{[V]} |TRNN(v_j)| \delta_{y}^j + (|TRNN(v_j)| - m_i) \delta_{y}^j \quad (8.13)$$

On the other hand, the number of $|V|-i$ nodes to the right of $v_i$ in FSA, are, however, moved as far as $\delta_{x}^i$ (or $\delta_{y}^i$) and $\phi_{x}^i$ (or $\phi_{y}^i$) respectively in the two directions. The entire adjustment for relocating node $v_j$ cumulates $(|V|-i) \delta_{x}^i$ and $(|V|-i) \phi_{y}^i$. According to (8.11) and (8.12), we therefore have:

$$f^{FSA}(v) = \sum_{i=1}^{[V]} (m_i |V|-i) \delta_{x}^i + m_i |V|-i) \phi_{x}^i$$

$$+ (|TRNN(v_j)| - m_i) |V|-i) \delta_{y}^j + (|TRNN(v_j)| - m_i) |V|-i) \phi_{y}^j \quad (8.14)$$

In both (8.13) and (8.14), where
Clearly, \(|\text{TRNS}(v_j)| \leq |V|-1\) hold, and from (8.13) and (8.14), we have \(f^\text{FTA}(v_i) \leq f^\text{FSA}(v_i)\).

**Theorem 8.2** Given a graph layout with overlapping nodes, the adjusted graph layout by FTA is more compact than one by FSA.

**Proof.** We use the *Cost Function* of Definition 5 to show the above claim. The smaller the value of the *Cost Function*, the more compact the layout. In other words, we need to show \(f^\text{FSA}(v_i) \leq f^\text{FTA}(v_i)\). The *Cost Function* is determined by both the force applied on a node and the number of nodes affected by this transferred force. From Lemmas 8.1 and 8.2, it is easy to see the result. Actually, we have \(f^\text{FSA}(v_i) < f^\text{FTA}(v_i)\) in most cases. It is only in special cases such as two overlapping nodes with the same \(x\) or \(y\) centre coordinates, that the equation \(f^\text{FSA}(v_i) = f^\text{FTA}(v_i)\) should hold.

We show the correctness and complexity of FTA by the Theorems 8.3 and 8.4.

**Theorem 8.3** Given a graph layout with overlapping nodes, FTA ensures that the adjusted graph layout has no overlapping nodes.

**Proof.** For any two nodes \(v_i\) and \(u_j\) on the same side of the *Seed Node* \(v_s\) in a graph, we need to show \(NS(v_i) = \phi\) or \(NS(u_j) = \phi\). With an adjustment, suppose that the original and new \(x\) coordinates of the nodes are \(x_i^v, x_j^v, x_i^w, x_j^w\), respectively, and \(x_i^w \geq x_i^v\), then we have

\[
x_i^w - x_i^v = x_i^w - x_i^0 + \sum_{n=1 \text{ or } s}^{j} \sum_{n=1 \text{ or } s}^{j} \delta_{mn}^x - \left(x_i^0 - x_i^0 + \sum_{n=1 \text{ or } s}^{j} \sum_{n=1 \text{ or } s}^{j} \delta_{mn}^x\right)
\]

\[
= x_i^w - x_i^0 + \sum_{n=1 \text{ or } s}^{j} \sum_{n=1 \text{ or } s}^{j} \delta_{mn}^x
\]

\[
\geq x_i^w - x_i^0 + \delta_{mn}^x
\]

\[
=x_i^w - x_i^0 \quad \left| x_i^0 - x_i^0 \right| + \frac{w_i + w_j}{2} - \frac{w_i + w_j}{2}
\]

Similarly, for \(y_i^v \geq y_i^w\), thus \(y_i^w - y_i^v \geq \frac{h_i + h_j}{2}\).

That is, the forces applied to these two nodes can separate them either in \(x\) or \(y\) direction with a distance of either \(x_i^w - x_i^v \geq \frac{w_i + w_j}{2}\) or \(y_i^w - y_i^v \geq \frac{h_i + h_j}{2}\). If nodes \(v_i\) and \(u_j\) are on different sides of *Seed Node* \(v_s\), such as the left and right sides, we
also can give a similar proof. In conclusion, we have $NS(v_i) = \emptyset$ or $NS(u_j) = \emptyset$.

**Theorem 8.4** The time complexity of FTA is $O(n^2)$.

**Proof.** As previously stated, the algorithm contains four scans. Here we look at the right scan again. The right scan starts by obtaining both the *Right Neighbor nodes* and *Transfer Right Neighbor* of every node on the right side of *Seed Node*. Obviously, the running time of the former is bounded by $O(n^2)$ since there are two loops in the pseudo-code in the previous section. The worst-case time complexity of the latter is also $O(n^2)$. By means of Theorem 8.1, the computational complexity of the *Transfer Right Neighbor* of each node is greatly reduced. The *While* statement in the right scan and the updating of the *Transfer Right Neighbor* of the next adjusted node will require $O(n^2)$ computational time at most. As such, the time complexity of FTA takes $O(n^2)$ in the worst case.

FTA cannot ensure to preserve the “orthogonal ordering” of the mental map model after an adjustment [Lai 1993; Misue et al. 1995]. The forces are only restrictedly transferred between neighbor nodes, leading to maintaining the relative positions among them. However, they cannot ensure that the relative positions between neighbored and non-neighbored nodes are preserved. For this reason, a slightly varied FTA is provided to preserve the “orthogonal ordering” mental map. This is accomplished by equivalently considering all the nodes in a graph. More precisely, a transferred force on a node causes all the successive node of this node in the scan direction to move identical distances, no matter whether they are neighbor nodes or not. As a consequence, by preserving the “orthogonal ordering”, this variant of FTA will produce a less compact graph layout than the original FTA, but still compact than FSA due to the reason presented in Lemma 8.1.

**8.6 Empirical Evaluation**

Both FTA and FSA have been implemented in PGD using the Java programming language as described in Chapter 3 (see Figure 3.6). In this section, we compare FTA with FSA by applying them to a set of arbitrary graphs, some of which are similar to those in [Marriott 2003], and discuss some features. First of all, a typical example using FTA and FSA is shown in Figure 8.5.
(a) An overlapping graph

(b) Layout adjusted by FTA

(c) Layout adjusted by FSA

Figure 8.5: An example of an overlapping graph adjusted by FTA and FSA

Figure 8.6 illustrates different resulting layouts for the same graph by FTA with various *Seed Nodes*. In particular, Figure 8.6(b) uses the leftmost node and Figure 8.6(c) selects the centre node as the *Seed Node*. To find the best candidate for a *Seed Node*, we should try every node to calculate the values of the *Cost Function* and select the minimum one.

Within our prototype system, users can alternatively select the *Seed Node* for an adjustment. This facility makes a faster layout than using the *Cost Function*. 
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

Figure 8.6: Initial and adjustment layouts for Graph 1

When selecting the central position as the dummy Seed Node for the graph in Figure 8.7 (a), for example, Figure 8.7 (b) shows the layout result. This layout is better than that in Figure 8.7(c) with the leftmost node as the Seed Node. This is because only the two nodes in Figure 8.7(b) had been moved to the left or right. In contrast, three nodes in Figure 8.7(c) had been repositioned to the right. More importantly, Figure 8.7(c) preserves the symmetry of the original graph.

Figure 8.7: Initial and adjustment layouts for Graph 2

Figure 8.8 (b) shows that an applied force stops transferring from one cluster node to another by using FTA. The reason for this is that formula (8.5) is not satisfied in this instance. Two overlapping nodes are located at the bottom level of Graph 3 shown in Figure 8.8(a). When removing the overlap, the applied force on the second node cannot pass on to other following nodes on the right direction. It is for this reason that makes more compact the layout by FTA than that by FSA in this example.

Figures 8.9, 10 and 11 compare three adjustment examples by applying both algorithms to Graphs 4, 5 and 6, respectively.
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

Figure 8. 8: Initial and adjustment layouts for Graph 3

These layouts adjusted by FTA are obviously more compact than those by FSA. Furthermore, their occupying areas by FTA are smaller than those by FSA.

Figure 8. 9: Initial and adjustment layouts for Graph 4

Figure 8. 10: Initial and adjustment layouts for Graph 5
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

The biggest node in Graph 7 shown in Figure 8.12(a) overlaps with two other nodes so that FTA moves only this node as shown in Figure 12 (b). FSA, however, repositions all the nodes on the right side of the first node overlapping the biggest one.

Graph 8 in Figure 8.13 exemplifies that a layout adjustment may introduce edge-node intersections by using FSA, whereas FTA does not produce edge-node intersections for this graph. In some cases, FTA, however, may also generate edge-node intersections. Fortunately, this problem can be easily solved by the approach presented in [Lai and Eades 2002].

Graph 9 in Figure 14 is an X-shaped graph with the symmetry about both the x- and y-axis. Both the layouts by FTA and FSA are good, but the gaps between the nodes in Figure 8.14(c) are a little bigger than those in Figure 8.14(b).
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

Figure 8.13: Initial and adjustment layouts for Graph 8

Figure 8.14: Initial and adjustment layouts for Graph 9
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

In summary, we conclude by the above examples that graphs adjusted by FTA are more compact than those by FSA. Furthermore, we can also precisely measure the qualities of adjusted layouts using the Cost Function previously defined. The adjustment costs for the previous example layouts are reported in Table 8.1, where the last two rows list the values of the Cost Functions. We also illustrate them in Figure 8.15. Note that the leftmost nodes ($v_1$) of those graphs were chosen as the Seed Nodes for the calculations of the Cost Functions.

\[
\begin{array}{cccccccc}
\text{Graph} & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{Nodes} & 10 & 13 & 8 & 12 & 10 & 5 & 18 \\
\text{FTA} & 7 & 127 & 86 & 76 & 4 & 18 & 258 \\
\text{FSA} & 74 & 1318 & 457 & 399 & 89 & 85 & 1335 \\
\end{array}
\]

Table 8.1: Value of Cost Function $f(v_1)$

In [Marriott 2003], the authors pointed out that the ranking of methods in terms of CPU time taken to find an adjusted layout, from fastest to slowest is: Uniform Scaling, Force-Scan, and Constrained Optimization. Uniform Scaling and Force-Scan are considerably faster than the Constrained Optimization. Our Force-Transfer algorithm has the same time complexity as that of Force-Scan. This new approach can make an adjustment compact as well as faster.

![Figure 8.15: Comparison of values of Cost Function for Graph 3 - Graph 9](image-url)
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

8.7 Applications

Current graph drawing systems usually employ static layout algorithms. Although static layout is useful in many applications, dynamic layout is essential in the following situations. Our algorithm can be effectively applied to all these situations.

*Incremental navigation in large graphs:* There is considerable interest in visualizing large graphs, such as computer networks, and linked hypertext documents. During incremental navigation of these large graphs [Huang 1998] - for example a Web graph - the users’ mental map should be preserved after changing the views of the graphs from one to another. Otherwise the users may spend extra cognitive efforts becoming familiar with the changes.

*Viewing dynamic data:* In many settings, graph diagrams are applied to visualizing dynamic systems. Dynamic graph layout could be an appropriate way to show the history of the data. Relative stability is necessary for such visualizations to reveal what is actually changing in the structure of the underlying data.

*Interactive graph editing:* Graph diagram systems usually support interactive editing. Dynamic layout algorithms could improve the usability of the interfaces in these systems.

8.8 Summary

Graphs where each node includes an amount of labels are often used in applications. A typical example of such graphs is UML diagrams in CASE tools. In order to make the textual label of each node readable in drawings of such graphs, it requires that there should be no overlapping nodes. This chapter has presented the Force-Transfer algorithm to remove overlapping nodes. The proposed approach employs a heuristic method to approximate a global optimal adjustment with the local minimal movement. Scanning from the *Seed Node*, this approach orthogonally transfers the minimum forces to only those nodes recursively overlapping with the node from where the forces start. We compare the Force-Transfer with the Force-Scan algorithm by mathematical proofs and experiments, and have shown that the Force-Transfer approach can generate better results. We conclude that FTA achieves an adjusted
CHAPTER 8. FORCE-TRANSFER: REMOVING OVERLAPPING NODES

graph layout with a good trade-off between a compact layout and running time. Our approach can be used not only to adjust graphs, but also to lay out non-overlapping objects such as windows and labels, by specifying a minimum gap between these objects. As an example, we already applied FTA to adjust the overlapping nodes and windows in MMD in the last chapter, and will show further results in the case study in the next chapter.
Chapter 9

A Case Study: Visual Web Navigation

In the previous chapters, we have presented all our techniques and approaches in relation to the proposed framework in Figure 3.3 in Chapter 3. Using an example of Web graph navigation, we will thoroughly demonstrate the effectiveness of these proposed algorithms and approaches in this chapter.

Part of the content of this chapter has been published in [Huang and Lai 2002].

9.1 Introduction

For Web navigation and exploration purposes, a graph can be used to represent the Web structure. In this graph, nodes represent Web artifacts such as the WWW, Web sites, Web pages, and even keywords in such Web pages, while edges indicate links that are between WWW sites, within Web sites, outside Web sites, and within Web pages, etc. This graph is called a Web graph. The structure of the Web is extremely complex, and so is its representation in the Web graph. In particular, navigation of the Web is characterized by the following:

- The size of the entire Web is so huge that the Web graph is a large and partially unknown graph [Huang and Eades 1998; Huang et al. 1998; Huang 1998].
- The information on the Web is hierarchically organized.
- The Web graph layout should be dynamically adjusted when users explore the Web.

Although several visualization techniques for coping with large graphs, such as Pad++ [Hollan 1994] and SemNet [Fairchild 1988], are available, they are not very suitable for the Web graph due to the above characteristics of the Web. Huang et al. [Huang and Eades 1998] proposed an on-line exploratory visualization approach,
which provides a major departure from traditional site-mapping methods. Such an approach does not pre-define the geometrical structure of a specific Web site (a part of cyberspace), but calculates incrementally and maintains a small visualization of a subset of cyberspace on-line, which is based on the change of users’ focus. In other words, a sequence of Web sub-graphs is automatically displayed with smooth animation based on the users’ orientation. This feature enables the users to logically explore the entire cyberspace without knowing the whole structure of cyberspace. This paradigm is mainly accomplished by an online graph drawing algorithm. The main idea of this technique is that users can explore a huge graph by changing logical frames consisting of a sequence of small sub-graphs. The definition of logical frames is as follows. Given a queue of the focus nodes $Q = (v_1, v_2, ..., v_s)$ of a graph $G = (V, E)$, the sub-graph of $G$ induced by a union of $N(v_1), N(v_2), ..., N(v_s)$ is called logical frames $F = (G', Q)$, with $G' = (V', E')$ where

$$V' = \bigcup_{i=1}^{s} N(v_i), \text{ and } E' = \{(u,v) \in E \mid u,v \in V'\}$$

where the distance-$d$ neighborhood $N(v_i)$ of node $v_i$ is the sub-graph of $G$ induced by a set of nodes whose graph-theoretic distance from $v_i$ is at most $d$. Formally, we have:

$$N(v_i) = \{u \mid D(v_i,u) \leq d \wedge v_i \in V \wedge v_i \neq u \wedge d \in \mathbb{R}^+\}$$

where the distance $d$ is a non-negative integer, and $d=1$ in the implementation of Huang et al.’s model.

The online graph model comprises a sequence of logical frames $F_1 = (G_1, Q_1)$, $F_2 = (G_2, Q_2), ...$, which is the explorative path of a Web graph. Successive frames may differ only by a few nodes. The sequence of logical frames represents the series of sub-graphs viewed by the users, and is determined by the interaction of the users who can change the focus and thereby decide which the new logical frame is to be displayed.

The remainder of this chapter starts by regarding the WWW as a hypergraph, and then describes models for WWW exploration. Finally, we illustrate an example of a Web graph.
9.2 The WWW as a Hypergraph

In this section, we formalize the view of the WWW as a hypergraph, based on the Definition 7.4.

A hypergraph \( G = (G', T') \) consists of a simple graph \( G' = (B \cup S, E_G) \) and a tree \( T' = (B \cup S, E_T) \) [Sander 1996]. The set \( B \) contains the leaves of \( T' \) which are called base nodes, and the set \( S \) contains the inner nodes of \( T' \), which are called abstract nodes or clusters. Given a node \( v \in (B \cup S) \), it has a coordinate \((i, j)\), denoted by \( v_{ij} \), which represents its position in \( G' \) and \( T' \), namely the integer \( i \) is the ordered number of a node in \( G' \) within the same height of \( T' \), and the integer \( j \) is the height of \( T' \). Obviously, \( j = 0 \) corresponds to the root of \( T' \), \( j = 1 \), the height 1 of \( T' \), and so on.

In a hypergraph, \( G' \) is a connective relation between base nodes and abstract nodes. The tree \( T' \) represents a nesting relation, while abstract nodes may contain other abstract nodes or base nodes.

We can consider the underlying structure of the WWW as a hypergraph, in which base nodes represent the WWW, sites, pages, and hyperlinks, etc.

Connectivity relation \( G' \)

Nesting relation \( T' \)
Abstract nodes can summarize those base nodes at different levels, i.e. \( B \cup S = \{ \text{WWW, site, page, hyperlink}, \ldots \} \), and abstract edges represent one or more edges between lower-level nodes.

With this representation, the WWW becomes a large graph containing several hundred million nodes and a few billion edges. Abstract nodes can be selected and opened to display, and they can also be automatically filtered. We use MMD described in Chapter 7, to view huge graphs at various levels of abstraction. Figure 9.1 shows part of the view of a site as a hypergraph.

### 9.3 The WWW Exploration Model

We extend the online graph model of Huang et al [Huang and Eades 1998] to three dimensions. In particular, nodes with the same abstract level \( j \) are drawn on a window, which corresponds to the original online graph model, while the \( z- \)
coordinate represents different abstract levels, i.e. the tree view at a height $j$ (see Figure 9.1). This makes possible multilevel navigation and exploration.

A sequence of logical frames consisting mainly of the online graph model is a subset of a graph that includes all possible sub-graphs of a graph $G$ induced by the union of neighborhood sets. Formally, we have

$$F_{ij} \subseteq \bigcup_{m,j=1,n,k=0}^{x,t} (G'_{mn}, Q_{lk})$$

where $t$ denotes the depth of $T'$, namely $t = \max(T'_{\text{height}})$, and the definitions of $G'_{mn}$ and $Q_{lk}$ are similar to the original ones presented previously.

The exploration path of a graph $G$ unifies a sequence of logical frames, i.e. $P= \bigcup_{i=m, j=n}^{s,t} F_{ij}$ where $m$ and $n$ are non-negative integers, meaning that the exploration begins with a node $v_{lk}$, and probably ends with a node $v_{st}$.

Based on the model defined above, we can roughly classify an exploration path into two patterns:

- **Horizontal Exploration**: users explore the Web artifacts located at the same abstract level of $T'$, i.e. $P_h = \bigcup_{m \leq n, j=k} F_{ij}$, where $m$, $n$ and $k$ are non-negative integers with $m<n$ and $0 \leq k \leq t$.

- **Vertical Exploration**: users explore different abstract levels of the Web artifacts, which can be further sub-classified into “top-down” and “bottom-up”, i.e. $P_v = \bigcup_{m \leq l \leq n, j \leq k} F_{ij}$, where $l$ and $k$ have the same value ranges as those of the above $m$ and $n$.

Within the vertical exploration, users can open or close abstract nodes in order to change the focus nodes along the tree $T'$. While exploring nodes from top to bottom in such a tree, they are shown from a more abstract level to a lesser one, following the Overview+Detail [ Card 1999] display. On the other hand, the Focus+Context [Hara, Lamping et al. 1995] display will be used instead while exploring nodes from bottom to top in the tree.
Summarily, following either the horizontal or vertical exploration model, users can explore a Web graph in different directions as well as at various levels of detail.

In order to visualize a Web site in the form of a hypergraph, the first question is, however, how to obtain such a hypergraph. In fact, the approaches presented in Chapters 5 and 6 can be applied to cluster a Web graph in order to generate a hypergraph. The following provides an example using the framework of graph visualization described in Chapter 3.

### 9.4 An Example

Using part of the Web site of Swinburne University of Technology as an example, we systematically illustrate our proposed algorithms and approaches including \textit{NodeRank}, SMK, MMD, and the Force-Transfer algorithm in this section.

Figure 9.2 shows a navigation example of a Web site. This system has the following salient features:

- It potentially supports the incremental exploration of Web sites
- The unimportant nodes and their associated edges can be filtered to improve the readability if needed
- The horizontal and vertical exploration models are provided, which enable users to navigate the Web sites both broadly and deeply

The interface of the system is shown in Figure 9.2, with the Web graph on the left, and the display of a Web page on the right. Except for some abstract nodes, each node in the Web graph is linked to a URL. For example, the nodes with labels www.swin.edu.au and www.yahoo.com.au are directly linked to the corresponding Web page contents respectively, as shown in the figure.
Figure 9.2: Navigation of a Web site
The process of the example shown in Figure 9.2 includes the following steps.

- **Data Extraction**

This example was extracted from the Web site of Swinburne University of Technology by the WebCrawler software described in Chapter 3. The software accepts a starting URL address as well as the exploration depth as its input, and then analyzes the hyperlinks among Web pages. A URL text file, which contains all the extracted URL addresses of two hyperlinked Web pages, is eventually produced for the following investigation.

![Diagram](attachment:image.png)

**Figure 9.3: Process of an example of visual Web navigation**

- **Filtering**

A filtering algorithm is applied directly to the above URL text file, or to a Web graph generated from the URL text file, where a node presents a Web page, and an edge
CHAPTER 9. A CASE STUDY: VISUAL WEB NAVIGATION

indicates a hyperlink between two Web pages. By specifying a threshold, we remove some “noise” nodes, or less important nodes. Figure 9.4 is an example. More examples are shown in Figure 4.8 in Chapter 4.

Figure 9.4: An example of filtering
CHAPTER 9. A CASE STUDY: VISUAL WEB NAVIGATION

- Clustering

After filtering, SMK, the clustering algorithm, is applied to the filtered Web graph. For simplicity, the highest abstract level of the clustered Web graph contains only two nodes as shown in Figure 9.5(a). This is achieved by increasing the parameter $\delta$ of the algorithm for finding seed nodes presented in Figure 5.1 in Chapter 5. With two expanded abstract nodes from Figure 9.5(a), Figure 9.5 (b) details all their children nodes and corresponding edges in the second abstract level of the Web graph.

![Figure 9.5: A clustered Web graph](image)

With the filtered and clustered Web graph, users can explore it based on the following models.
CHAPTER 9. A CASE STUDY: VISUAL WEB NAVIGATION

- Vertical Exploration

Vertical exploration: As described before, users can open/close parent or child nodes of the current abstract node. By opening an abstract node, the users can open a new overlay window which displays part of the logical frames of the abstract node.

Starting from the root nodes or the highest abstract level of the clustered Web graph, the user may click on the node with the label “www.it.swin.edu.au” shown in Figure 9.5 (a), in order to display its children nodes shown in Figure 9.6(a). Suppose that the user continues to explore one of its children and abstract nodes labeled “current.htm”, the resulting layout is illustrated in Figure 9.6 (b). Finally, the node “programs.htm” is opened in the same manner as shown in Figures 9.6 (b) and (c). In other words, along with the abstract tree $T$, the exploration processes from the root to the next abstraction level. This leads that the display windows are automatically adjusted in accordance with the number of children nodes using the algorithm presented in Chapter 7.

(a) The Web graph with the current focus window: it.swin.edu.au, and the current node: current.htm
CHAPTER 9. A CASE STUDY: VISUAL WEB NAVIGATION

(b) The Web graph with the opened node with label "current.htm"

(c) The Web graph with the opened nodes with labels "programs.htm" and "current.htm"

Figure 9.6: Vertical Exploration of the Web site

- **Horizontal Exploration**

In the case of the horizontal exploration, the user’s interaction may move from one logical frame to another by choosing a focus node with the same abstraction level. The window will thus become bigger and bigger until it completely fills the whole screen. Both the layout of a logical frame and the expansion direction of the window
clearly show the exploration direction of the Web hypergraph. For example, with a different highlight window, the users move the focus node “http://www.it.swin.edu.au” in Figure 9.6(a) to another node “http://www.swin.edu.au” in Figure 9.7. Note that the two nodes have the same abstraction level.

Figure 9.7: Horizontal exploration of the Web site

- **Dynamic Adjustment**

During the exploration, there may be overlapping windows, nodes or labels. In order to improve the readability of a layout and to keep users’ mental map intact, we need to adjust the layout. Figure 9.8 shows the process of an adjustment by applying our algorithm FTA described in Chapter 8.
CHAPTER 9. A CASE STUDY: VISUAL WEB NAVIGATION

(a) The Web graph with overlapping labels

(b) The choice of applying the Force-Transfer algorithm
CHAPTER 9. A CASE STUDY: VISUAL WEB NAVIGATION

In this chapter, we have illustrated the navigation of a Web graph as an example of the use of our framework. Starting with regarding the Web site as a graph, and more specially, a hypergraph, we have described two models of Web exploration. A case study of Web navigation was then provided in detail with illustrative examples.
Chapter 10

Conclusion and Future Work

10.1 Conclusion

In this thesis, we have presented a framework for graph visualization applications and developed the algorithms and approaches related to such a framework.

Starting with identifying the characteristics of graph visualization applications, we proposed a framework for describing general stages of such applications.

Within the framework, we provided a generic approach to modelling a graph from given information source, i.e. a set of objects and their attributes. With the purpose of graphical representation of these objects and their relations, the underlying issue is whether a relation between two objects is strong enough to build an edge representing such a relation in graph visualization. The basic idea of our approach is that building the relations is based on the degree of similarity between the attribute values of two objects, which in turn comes from the combination of node attributes.

We investigated two approaches, Filtering and Clustering, to dealing with large graph layouts.

A structure-based approach was proposed for filtering. The key idea of this approach is how to calculate the importance of a node in a graph. Two methods for the calculation of such importance values were provided on the basis of the approaches in social network analysis. By ranking nodes in terms of their importance values, a graph can be globally filtered or locally suppressed with either Global filtering or Fisheye view models. As a structure-based method, our approach has wide applications in comparison with others.
CHAPTER 10. CONCLUSION AND FUTURE WORK

We developed two approaches to clustering a graph for the layout. Both approaches take advantage of the structural patterns of a graph rather than the semantic contents of nodes, since the goal of clustering is for the layout. After identifying relatively high degrees of nodes as initial members of clusters, two approaches find the rest of the available nodes as new members of the clusters using different strategies. The first approach called KNN is built on an affinity function and the \( k \)-nearest neighbour. The second approach models the linkage of a graph into a similarity matrix, and then groups the most similar nodes into a cluster within such a matrix. Two approaches have effectively demonstrated achievement of clustering graphs by examples. These approaches have their advantages and disadvantages as discussed in the previous chapters.

For a dynamic graph drawing, or the layout of a labelled graph, there is a problem of removing overlapping nodes. In these two scenarios, the minimal adjustments of an original graph layout are required to improve readability and to maintain the mental map. Although there are already several algorithms to resolve this problem, our new approach called Force-Transfer algorithm was presented in Chapter 8. The basic idea of this algorithm is to combine the local minima with global minima, in order to achieve an approximate minimal adjustment in reasonable time. We validated our algorithm by both theoretical proof and experimental examples. In terms of compact adjustments and time complexity, our approach is a good candidate for the balance choice.

An approach to the layout of clustered graphs was also developed, with the additional purpose of navigation and exploration.

Finally, we conducted a case study to systematically demonstrate our work.

10.2 Future Work

Our experiments have demonstrated that our algorithms and approaches are so effective that our research objects have been reached. However, this research still requires further investigation of the following aspects:
CHAPTER 10. CONCLUSION AND FUTURE WORK

1. A weighted graph: Note that our algorithms and approaches are confined to an undirected graph. We need to investigate the likelihood of extension of the proposed algorithms and approaches into other types of graphs, such as weighted graphs or digraph graphs. In particular, we proposed an approach to measuring the similarity degrees of two nodes in a graph in Chapter 6. This graph can be viewed as a special case of a weighted graph. For the metric to measure the similarity degree among nodes in a graph, it is possible to extend such a metric to a weighted graph. For this reason, we need to reconstruct the underpinning similarity degree between two nodes.

2. An attributed graph: We mention an attributed graph in Chapter 3. How to filter and layout an attributed graph is another question, with consideration of multiple relations between two nodes. Our work can be extended to handle an attributed graph.

3. A semantic graph: We only consider the structure of a graph in our current work. In our future work, we may combine the semantic nodes with their structure. In other words, a hybrid approach should be employed in some situations.


22. Dogrusöz, U., Madden, B. and Madden, P., Circular Layout in the Graph Layout Toolkit. in *4th International Symposium on Graph Drawing*, (Berkeley, California USA, 1996), 92-100.

23. Dongen, S.V., Graph Clustering by Flow Simulation *Mathematics and Computer Science*, University of Utrecht (research carried out at Centre for Mathematics and Computer Science, CWI), Utrecht, 2000, 169.

25. Eades, P. and Feng, Q.-W., Multilevel Visualization of Clustered Graphs. in 4th International Symposium on Graph Drawing, (Berkeley, California USA, 1996), Springer Verlag, 101-112.
40. Flake, G., Lawrence, S., Giles, C.L., Efficient Identification of Web Communities. in Proceeding of the Sixth International Conference on Knowledge Discovery and Data Mining, (2002), 150-160.


71. Huang, M.L., Information Visualization of Attributed Relational Data. in *Australian Symposium on Information Visualization*, (Sydney, Australia, 2001), Conferences in Research and Practice in Information Technology, 143-150.


77. Huang, X.D., Wei, L., Clusters in the Web Graph by Using Local Search. in *Proceedings of the International Conference on Internet Computing*, (Las Vegas, USA, 2003), 83-89.

78. Huang, X.D., Wei, L., Force-Transfer: A New Approach to Remove Node Overlapping in Graph Layout. in *Australian Computer Science Communications*, (2003), 349-358.

79. Huang, X.D. and Wei, L., Identification of Clusters in the Web Graph based on Link Topology. in *Proceedings of the 7th International Database*
80. Huang, X.D. and Wei, L., Identifying Clusters in the Web Graph by Using Local Search. in Proceedings of the International Conference on Internet Computing, (Las Vegas, USA, 2003), 83-89.


82. Huang, X.D. and Wei, L., WWW as a Hypergraph: Exploration and Layout. in Proceedings of the International Conference on Internet Computing, (Las Vegas, USA, 2002), 615-621.


111. Lawrence, P., Rajeev, M., and Terry, W., The pagerank citation ranking: Bringing order to the web, Stanford University, 1998.


Bibliography

126. Matula, D.W., Graph theoretic techniques for cluster analysis algorithms. in Ryzin, V. ed. *Classification and Clustering*, 1987, 95-129.
129. Meyer, B., Self-Organizing Graphs A Neural Network Perspective of Graph Layout. in *International Symposium on Graph Drawing*, (1998), Springer LNCS.
132. Müller, M.D., Graph Layout Adjustment Strategies. in *Symposium On Graph Drawing, GD’95*, (Passau Germany, 1995), Springer-Verlag, 487-499.


Bibliography


What is the UML

The Unified Modelling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artefacts of a software-intensive system”

---- Grady Booch, etc.
UML diagrams
● Each view is a projection into the organization and structure of the system focused on a particular aspect of the system.
Architecture: Use Case View

- Encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers.
- Static aspects:
  - Use case diagrams
- Dynamic aspects:
  - Interaction diagrams
  - Statechart diagrams
  - Activity diagrams
Architecture: Design View

- The classes, interfaces, and collaborations that form the vocabulary of the problem and its solution.
- Supports that functional requirements of the system.
- **Static aspects**: Class diagrams, Object diagrams
- **Dynamic aspects**: Interaction diagrams, Statechart diagrams, Activity diagrams
Architecture: Process View

- Threads and processes that form the system’s concurrency and synchronization mechanisms.
- Addresses performance, scalability, and throughput.
- **Static aspects**: Class diagrams (focus: active classes), Object diagrams
- **Dynamic aspects**: Interaction diagrams, Statechart diagrams, Activity diagrams
Architecture: Implementation View

- The components and files that are used to assemble and release the physical system.
- Primarily addresses the configuration management of the system’s releases.
- **Static aspects**: Component diagrams
- **Dynamic aspects**: Interaction diagrams, Statechart diagrams, Activity diagrams
Architecture: Deployment View

- Encompasses the nodes that form the system’s hardware topology on which the system executes.
- Addresses the distribution, delivery, and installation of the parts that make up the physical system.
- **Static aspects**: Deployment diagrams
- **Dynamic aspects**: Interaction diagrams, Statechart diagrams, Activity diagrams
Object-Oriented Concepts

● Five necessary characteristics
  • information hiding
  • data abstraction
  • dynamic binding
  • inheritance
  • polymorphism
Information Hiding

- Parnas [1972] articulated importance of information hiding during system design & decomposition.
- Also known as “encapsulation”
  - Interface to class captures external view
  - Implementation details hidden from external entities
    » private attributes
    » methods
    » representation
Information Hiding

Objects can contain other objects
Benefits of Encapsulation

- Reduces propagation of side effects when changes occur.
- Merging data structures and operations facilitates component reuse.
- Interfaces among encapsulated objects are simplified (messages) -- interfacing is simplified and coupling is reduced.
Data Abstraction

- Usually coupled with the foregoing principle (information hiding)
- Denotes essential characteristics of an object -- What makes an object a member of a class?
- Enables a designer to represent a data object at varying levels of detail
- Specification of data object in the context of those operations that can be applied to it.
Dynamic Binding

- Binding of an instance to a characteristic
- Determined by the time during program compilation when this occurs:
  - composition time - (really!) static
  - compilation time - static
  - execution time - dynamic
Inheritance

- A “kind of” hierarchy of abstractions
- Defines a relationship among classes wherein one class (subclass) shares the structure or behavior defined in one or more other classes (superclass(es)).
  - Single: one other class
  - Multiple: more than one other classes
- A subclass may augment or redefine existing structure/behavior of the superclass(es)
Polymorphism

- A concept in type theory in which a single name may denote many different things:
  - “+” means the “same thing” for reals and integers
  - “+” is implemented differently for reals and integers

- Polymorphism is a consequence of the interaction of inheritance and dynamic binding:
  - a (sub)class inherits the name of an operation
  - the binding of a method to that name does not occur until execution
Object-Oriented Basics: Objects

- A model of anything, e.g. a chair, represents its salient characteristics — those characteristics that facilitate solving the problem for which the model is being built.
Object-Oriented Basics: Attributes

- An attribute of an object (class) is a value of a specific type that encodes information about the object under consideration.
Object-Oriented Basics: Operations, Methods, and Services

- The operations, methods, or services of an object are the processes (algorithms) that act on the data encapsulated by the object.

<table>
<thead>
<tr>
<th>object: chair</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost</td>
</tr>
<tr>
<td>dimensions</td>
</tr>
<tr>
<td>weight</td>
</tr>
<tr>
<td>location</td>
</tr>
<tr>
<td>color</td>
</tr>
<tr>
<td>buy</td>
</tr>
<tr>
<td>sell</td>
</tr>
<tr>
<td>weigh</td>
</tr>
<tr>
<td>move</td>
</tr>
</tbody>
</table>
Object-Oriented Basics: Messages

- Messages are the mechanism by which objects (classes) interact.

**object: sender**

<table>
<thead>
<tr>
<th>attributes:</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>operations:</th>
</tr>
</thead>
</table>

**object: receiver**

<table>
<thead>
<tr>
<th>attributes:</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>operations:</th>
</tr>
</thead>
</table>

**message:** [sender, operation, parameters]

**message:** [receiver, return value(s)]
E.g. A chair is a member of (instance of) a set of similar objects (class) that we call furniture.
Objects inherit from the class all attributes and operations.
Associations

- In the same sense that classes correspond to nouns, associations correspond to verbs.
- They express the relationship between classes.
- There are instances of associations, just as there are instances of classes.
- Instances of a classes are called objects;
- Instances of associations are called links in UML.
Class A and B are associated if

- an object of class A sends a message to an object of class B
- an object of class A creates an object of class B
- an object of class A has an attribute whose values are objects of class B or collections of objects of class B
- an object of class A receives a message with an object of class B as argument

in short, if some object of class A has to know about some object of class B
Simple association between classes

- One annotation which is often used early is the multiplicity of an association
- This is so fundamental that we will spend some time thinking about it.
Example

- Every copy is associated by *is a copy of* with just one book. We showed a 1 at the Book end of the association.
- On the other hand, there may be any number of copies of a given book in our system. So the multiplicity on the Copy end is 1..*.
Multiplicity

We can specify:
- an exact number simply by writing it
- a range of numbers using two dots between a pair of numbers
- an arbitrary, unspecified number using *

Loosely, you can think of UML’s * as an infinity sign, so the multiplicity 1 .. * expresses that number of copies can be anything between 1 and infinity.
Aggregation and composition

- Aggregation and composition are kinds of association:
  - Instead of just showing that two classes are associated we may choose to show more about what kind of association this is

- Aggregation and composition are both ways of recording that an object of one class
  
  is part of

  an object of another class.
Module is a part of an HonoursCourse

- The notation with open diamond, denotes aggregation, which is more general way of denoting a part-whole relationship in UML
• **Aggregation is essentially a conceptual notion:**
  - seeing an aggregation in a class model should help you to understand the relationships between the classes at an informal level
    » BUT does not give you any more formal information about how they must be implemented or what you can do with them
Composition

● Composition is a special kind of aggregation which does impose some further restrictions.

● In composition association, the whole *strongly owns* its parts
  • If the whole object is copied or deleted, its parts are copied or deleted with it

● The multiplicity at the whole end of a composition association must be 1 or 0..1
  • A part cannot be part of more than one whole by composition
Example

Noughts and Crosses
(Tic-Tac-Toe)

- Composition is shown just as aggregation is, except that diamond is filled in
Roles

- Often you can read an association name in both directions (‘is taking’, ’is taken by’)
- Sometimes, however, it is more readable to have separate names for the roles that the objects play in the association.
Association with no navigability

The diagram records that:

- For each object of class *Student* there are six objects of class *Module* which are associated with the *Student*;
- For each object of class *Module* there are some *Student* objects (the number of students is unspecified) associated with the *Module*.
Navigability

- We can put an arrow on one or both ends of the association line to represent that it is possible for messages to be sent in the direction of the arrow.

- We say that Module knows about Student, but not vice versa.
Qualified associations

- Occasionally it is helpful to give finer detail about an association than we have so far.
  - Square is identified relative to the board it’s on by attributes raw and column, each taking a value between 1 and 3.
Qualified composition

- In fact we can combine the qualified association notation with the other adornments on associations
- For example, we can add back the information that this particular association is a composition
Association classes

- Think which is both an association and a class, which is unsurprisingly called an \textit{association class}

\begin{itemize}
  \item Student \hspace{1cm} 1..* \hspace{1cm} \text{is taking} \hspace{1cm} \text{6} \hspace{1cm} \text{Module}
  \item is taking
  \item mark : int
\end{itemize}
Avoiding an association class

- Student
  - 1..* is taking
  - 6
  - 1

- Module
  - 1

- Mark
  - mark : int
  - 6
  - 1..*
Generalization

- Important relationship which may exist between classes is generalization

An object of a specialized class can be substituted for an object of a more general class in any context which expects a member of the more general class, but not the other way round.
Generalisation and inheritance

- Objects are members of classes which define attribute types and operations
- Classes may be arranged in a class hierarchy where one class (a super-class) is a generalisation of one or more other classes (sub-classes)
- A sub-class inherits the attributes and operations from its super class and may add new methods or attributes of its own
- Generalisation in the UML is implemented as inheritance in OO programming languages
A generalisation hierarchy

- Employee
  - Manager
    - budgetsControlled
    - dateAppointed
  - Programmer
    - project
    - progLanguage
  - Project Manager
    - projects
  - Dept. Manager
    - dept
  - Strategic Manager
    - responsibilities
Advantages of inheritance

- It is an abstraction mechanism which may be used to classify entities
- It is a reuse mechanism at both the design and the programming level
- The inheritance graph is a source of organisational knowledge about domains and systems
Appendix

In this appendix, a number of experimental examples are shown by using our proposed algorithms and approaches including NodeRank, KNN, SMK, FTA and MMD presented in this thesis.

1. Filtering Examples

Example 1
The filtered Web graph 1
The Filtered Web graph 1 and the corresponding Web content of one node with the yellow color

Example 2

A Web graph 2
APPENDIX

Example 3

A Web graph 3
The filtered Web graph 3

2. Clustering Examples

A clustering example for a Web graph

The following is part of a file describing the hyperlinks between two Web pages in a Web site (http://www.swin.edu.au), which was generated by the WebCrawler.

The following is part of the resulting clusters by using our algorithm, where “ROOT”. 0 and 1 represent the abstract level of the clustering.

+ROOT;+ROOT=0;http://www.swin.edu.au/~http://www.swinexpo.com

The following figures are based on the above results.
APPENDIX

A clustering example for a general graph

An example of a generated file is shown here, which specifies the resulting clusters of a hierarchically clustered graph, where the “ROOT”, 1 and 2 represent the abstract levels of clustering.

```
ROOT; ROOT==0;22_2427--0;1_15_173738
ROOT; ROOT==0;22_2427--0;18_21252628_38
0;1_15_173738==1;1415_173739--1;7_13
0;1_15_173738==1;1415_173739--1;1_6
0;1_15_173738==0;22_2427--1;1415_173739
0;1_15_173738==1;7_13--1;1_6
0;18_21252628_38==1;18_21--1;252628_36
0;18_21252628_38==1;252628_36--0;22_2427
1;1_6==2;1--2;4
1;1_6==2;2--2;4
1;1_6==2;3--2;4
1;1_6==2;5--2;4
1;1_6==2;6--2;4
1;1_6==2;7--2;4
1;1_6==2;1--2;2
1;1_6==2;3--2;2
1;1_6==2;5--2;2
1;1_6==2;7--2;2
1;1_6==2;8--2;2
1;1_6==2;11--2;2
1;1_6==2;12--2;2
```
A series of coarse graph layouts with different abstract levels are shown below on the basis of the above clustered results.
3. Examples of Comparison FTA with FSA

Several examples of random overlapping graphs and their adjusted graph layouts are shown below, together with the costs of such adjustments. Each example includes three figures. The first is an overlapping graph layout while the second shows the resulting layout by FTA, followed by an adjustment layout with FSA.

Example 1

Graph 1

Layout by FTA with $f^{FTA} = 168.08$  Layout by FSA with $f^{FSA} = 411.14$
Appendix

Example 2

Graph 2

Layout by FTA with $f_{FTA} = 186.39$

Layout by FSA with $f_{FSA} = 1309.02$
APPENDIX

Example 3

Graph 3

Layout by FTA with $f^{FTA} = 563.78$

Layout by FSA with $f^{FSA} = 1901.63$
APPENDIX

Example 4

Graph 4

Layout by FTA with $f^{FTA} = 276.32$
APPENDIX

Layout by FSA with $f_{\text{FSA}} = 745.26$

Example 5

Graph 5
Layout by FTA with $f^{FTA} = 183.14$

Layout by FSA with $f^{FSA} = 972.46$
APPENDIX

Example 6

Graph 6

Layout by FTA with $f^{FTA} = 7534.0$
APPENDIX

Layout by FSA with $f^{FSA} = 44771.16$ (This figure has been contracted)

Example 7

Graph 7

Layout by FTA with $f^{FTA} = 177.0$

Layout by FSA with $f^{FSA} = 760.63$