INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
GENERATING CUSTOMIZED LAYOUTS AUTOMATICALLY

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAII IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMMUNICATION & INFORMATION SCIENCES

AUGUST 1995

By

Xiaobo Wang

Dissertation Committee:

Isao Miyamoto, Chairperson
Jaw-Kai Wang
Kazuo Sugihara
David N. Chin
Larry N. Osborne
© Copyright 1995
by
Xiaobo Wang
All Rights Reserved
To
My Parent
I would like to thank Isao Miyamoto, my advisor, for his advice and long term support. I would also like to thank Kazuo Sugihara and David Chin for their useful suggestions.

I would like to express my thanks to Larry Osborn, Wang Zhang Kai, William Remus, and Art Lew for their guidance and interest in my work.

Thanks must be extended to my colleagues in Software Engineering Research Laboratory and Communication and Information Science Program, University of Hawaii: Ke Xiaongdong, Mistuguki Inaba, Keiichi Nemoto, Koji Takeda, Hai Hunag, Hu Bin, Ya Liu, and Douglas Wake. It has been an exciting experience to work with them. They also gave me interesting and useful comments. Hu Bin helped me implement the initial version of LYCA.

I would like to thank my wife, Ren Qing. Without her support, I could never have done this work.
Abstract

Most existing automatic layout techniques are designed to generate layouts that look pleasant to the eye by improving aesthetics of graphs. Aesthetics, however, do not reflect layout requirements derived from semantics, preference or individual situations. It is important for an automatic layout technique to generate customized layouts according to specific requirements given by the user or applications.

This thesis investigates how to generate customized layouts using selected layout algorithms. A key to this problem is to improve the expressive power of existing algorithms and integrate different techniques to deal with various layout requirements.

LYCA is a graph tool that uses incremental optimization algorithms to draw directed and undirected graphs. It integrates a constraint solver to process constraints. Compared with other works, LYCA has several distinctive features:

- The force-directed placement algorithm is improved to generate compact layouts for graphs with large vertices.
- A novel usage of the divide-and-conquer approach is introduced to generated structured layouts.
- The constraint solver and the layout algorithms are integrated in a simple and efficient way. In addition, the solver and layout algorithms cooperate to ensure layout quality.
- Different interface techniques are used to help the user diagnose layout problems and interact with the layout algorithms directly.

Those features provide a tight coupling of the user and the layout tool. Users can generate customized layouts with LYCA easily and flexibly.
# Table of Contents

Acknowledgments ......................................................... v
Abstract .................................................................. vi
List of Tables .............................................................. xi
List of Figures .............................................................. xii

1 Introduction ................................................................ 1
  1.1 Requirements ...................................................... 2
  1.2 The Problem ........................................................ 7
  1.3 Overview of the Dissertation ................................... 9

2 Related Work .......................................................... 10
  2.1 Layout Modes ...................................................... 10
  2.2 Layout Algorithms ............................................... 12
    2.2.1 Algorithmic Approach .................................... 12
    2.2.2 Declarative Approach ..................................... 15
    2.2.3 Integrated Approach ...................................... 17
  2.3 Constraint-Based Geometry Layout ........................... 18
  2.4 Other Approaches ............................................... 19
  2.5 Comparison of Different Approaches ....................... 20

3 Overview of LYCA .................................................... 22
  3.1 Architecture of LYCA ........................................... 22
  3.2 Data Structure and Graph Representation .................. 24
    3.2.1 Internal Representation .................................. 24
    3.2.2 External Representation .................................. 25

4 Layout Algorithms .................................................... 27
A.1.3 Menu Commands .............................................. 121
A.2 Constraint Language ........................................ 133
Bibliography ........................................................... 134
List of Tables

7.1 Experiment result for $G_1$ ................................................. 99
7.2 Experiment result for $G_2$ ................................................. 100
7.3 Experiment result for $G_3$ ................................................. 100
7.4 Experiment result for $G_4$ ................................................. 101
7.5 Experiment result for $G_5$ ................................................. 101
7.6 Result of the second experiment ......................................... 104
## List of Figures

1.1 Textual representation vs. pictorial representation ................. 1
1.2 Manual layout ................................................. 3
1.3 Automatic layouts ........................................... 4
1.4 Visual organization features .................................. 5
1.5 Different layout conventions (created with GraphEd) .............. 6
1.6 Incompatible aesthetics ......................................... 6
1.7 Layout requirements ........................................... 8

2.1 Example layout generated by force-directed placement ............ 13
2.2 Example layout of directed graph ................................ 15
2.3 Visually organized layout created by genetic algorithm .......... 16
2.4 The integrated approach of EDGE ............................... 17
2.5 Nondeterminism in an algorithmic method ........................ 18
2.6 Example layout generated by rule-based algorithm ............... 19

3.1 Configuration of LYCA .......................................... 22
3.2 Sample MERA graph ............................................. 25

4.1 Layouts created by force-directed placement ..................... 28
4.2 Distance between large vertices ................................ 29
4.3 Compact layout generated by LYCA (40 percent of actual size) ... 30
4.4 Another example of compact layout generated by LYCA (40 percent of actual size) ........................................... 31
4.5 Meta graph and meta layout ..................................... 32
5.16 Original layout .............................................. 71
5.17 Layout with constraints ..................................... 72
5.18 Improved layout of directed graph with constraints ........... 72
5.19 Layout that reflects zone and shape of subgraph ............... 74
5.20 Layout that reflects proximity ................................ 74
5.21 Layout without constraints ................................... 76
5.22 Layout created at time t’ ..................................... 76
5.23 Resulting layout ................................................. 77
5.24 Example layout with constraints ............................... 77
5.25 Unstructured layout ............................................. 78
5.26 Layouts that convey different information ..................... 78
5.27 An example layout of directed graph with constraints ........ 79
5.28 Another example layout with constraints ...................... 79

6.1 Animation ......................................................... 82
6.2 Inferior local minima and manual up-hill climbing ................ 83
6.3 Poor layout caused by barriers .................................. 83
6.4 Layout without barrier .......................................... 84
6.5 Poor layout caused by squeezing problem ....................... 84
6.6 Layout without squeezing problem ............................... 85
6.7 Layout monitor window ......................................... 86
6.8 Plots of aesthetics ................................................. 87
6.9 Plot of number of edge crossings ................................ 89
6.10 Animation and editing .......................................... 90
6.11 Parameterize force-directed placement algorithm ............ 93
6.12 Parameterize Sugiyama algorithm .............................. 94
6.13 Parameterize finetune algorithm ................................ 95
6.14 LYCA’s main window ........................................... 96

7.1 Test Graphs ..................................................... 98
7.2 Layouts of $G_1$ and $G_2$ ................................. 102
7.3 Layouts of $G_3$, $G_4$, and $G_5$ ...................... 103
7.4 Layout with overlapped vertices ......................... 105
7.5 Layout of graph $G_4$, created by spring algorithm .... 107
7.6 Layout of $G_5$, created by spring algorithm ........... 108
7.7 Layout of $G_4$, created by LYCA ...................... 109
7.8 Layout of $G_5$, created by LYCA ...................... 110

A.1 Layout buttons ............................................... 120
A.2 Rotate a graph .............................................. 120
A.3 Bend edges .................................................... 120
A.4 Straighten long path and adjust layout frame .......... 120
A.5 Define vertices and edges .................................. 121
A.6 Menu bar in LYCA’s main window ...................... 121
A.7 Load graph window ........................................ 122
A.8 Editing a vertex ............................................. 123
A.9 Display option for subgraph zone ...................... 125
A.10 Control buttons on monitor window ................... 125
A.11 Window for force-directed placement algorithm ....... 127
A.12 Window for Sugiyama algorithm ....................... 128
A.13 Window for the finetune algorithm ................... 129
A.14 Window for constraints ................................... 131
A.15 Before selecting “Solve constraint” command .......... 131
A.16 After selecting “Solve constraint” command .......... 132
Chapter 1

Introduction

Graphs serve as a fundamental data structure in computer science. They are also one of the best ways of presenting technical information pictorially. It is well known that pictures convey information more efficiently than text. Myers et al. [37] pointed out "Human information processing is clearly optimized for pictorial information, and pictures make the data easier to understand for the programmer." The example in Figure 1.1 is used in [46] to explain why a nicely drawn graph "is worth a thousand words." Two representations of a graph are given in Figure 1.1: one as a list of edges of the graph, the other as a drawing of the graph. From the drawing of the graph, the user can easily identify important properties of the graph, e.g., it is a binary tree with root at $K$ and depth of four. Although the textual description contains the same information, it is not obvious at first glance.

\begin{figure}[h]
\centering
\begin{tabular}{ll}
LIST OF EDGES \\
\hline
(K, B) & (K, C) \\
(D, H) & (B, E) \\
(C, F) & (C, G) \\
(F, J) & (G, A) \\
(E, I) & (B, D) \\
\end{tabular}
\end{figure}

A graph itself does not contain any information in its visual presentation. The task of positioning vertices and edges of a graph on the screen is called graph layout. Given a graph, there are an infinite number of ways to draw it. Most layouts of a
Graphs are misleading rather than informative. The goal of graph layout is hence to generate layouts that convey the meanings of the graphs clearly and quickly to users.

Graphs can be drawn manually. But manual layout is a tedious and error-prone process. For applications such as software maintenance tools, hundreds of graphs can be used to represent complex problems. In other applications, graphs may be changed in seconds, e.g., function-call graphs. Humans cannot draw hundreds of graphs within a reasonable time. It is also infeasible for humans to draw graphs that change constantly. An automatic method is the only solution for such applications.

The example in Figure 1.2 shows a manual layout used in [16]. Two automatically generated layouts of the same graph are shown in Figure 1.3. The automatic layouts not only look better than the manual layout, they are also generated much more efficiently.

### 1.1 Requirements

The requirements on automatic layouts must be clarified before automatic layout methods can be discussed. A classic and fundamental requirement on graph layouts is the aesthetic criteria. As summarized in [54]:

...in almost all data presentation applications, the usefulness of a graph depends on its readability, e.g., the capability of conveying the meaning of a diagram quickly and clearly. Readability issues are expressed by means of aesthetics, which can be formulated as optimization goals for the drawing algorithms...

Some common aesthetics are:

- a small number of edge crossings
- reflecting symmetry of a graph
- avoiding bends in edges
- even distribution of vertices
- a small area that a layout covers
Figure 1.2: Manual layout

Aesthetic criteria concern properties that make graphs pleasant to the eye. But they cannot deal with features related with the meaning of a graph. Instead, semantics of a graph must be expressed as layout constraints. A layout constraint defines certain spatial relation(s) that must hold in the resulting layout. With constraints, logic relations represented by a graph can be reflected as spatial relations in the drawing of the graph explicitly and consistently. For example, a critical path in a
Petri-net diagram should be drawn as a straight-line to highlight the dependencies among tasks.

Another requirement from users is to create visually organized layouts [10]. Studies on perceptual psychology revealed that a human's seeing process is an active one in which our view of the world is constructed both from information in the environment and from previously stored knowledge [45]. One aspect of the constructive process involves decomposing images into separate entities that are readily recognizable. The innate laws which humans use to organize scenes or images are described by the *Gestalt Laws* of perceptual psychology [24]. Graph layouts are not exempt from those laws. It was found that graphic users routinely use the grouping principles derived from the Gestalt Laws to organize and interpret images on screens [26] (see Figure 1.4). Furthermore, visual displays with inappropriate organizations often cause misunderstanding of the underlying problems [25, 31]. Therefore, a graph layout should be well-organized so it can be interpreted correctly by users.
A good summary of requirements on automatic layouts is given in [30]:

- **Readability.** A layout should be nice-looking and easy to read.

- **Conformance.** A layout should conform to the syntactic and semantic requirements given by an application. For example, different applications may require different representations of graphs, as shown in Figure 1.5. A layout tool should not impose a predefined conformance criteria on the user.

- **Controllability.** Users should be able to control automatic layouts in order to generate customized layouts.

- **Response time.** Layouts must be created efficiently to ensure reasonable response time.

To some extent, the second and third requirements are not exclusive. For example, to make layouts conform to semantic requirements, constraints should be supported to control automatic layouts.

Ideally, an automatic layout method should meet all the above requirements. This, unfortunately, is very difficult because many layout requirements lead to computationally intractable problems. For example, minimizing the number of edge-crossings is known as NP-hard [3]. Furthermore, different layout requirements may be competitive and incompatible. The two layouts in Figure 1.6 achieve the aesthetic
criteria of displaying symmetries and minimizing the number of edge-crossings, respectively. But it is impossible to optimize both aesthetic criteria for the graph in the same layout.

The layout problem becomes even more complicated when different requirements are combined together. It is in general infeasible to achieve all the above layout requirements at same time. Most layout algorithms focus on one or a few specific aspects of the layout problem. In addition, heuristics are often incorporated to balance competing requirements and trade off layout quality and time performance.
1.2 The Problem

Most automatic layout methods cited in [3] focus on the aesthetic issue and the efficiency issue. Such layout methods can generate nice-looking layouts efficiently. But they draw graphs according to a set of predefined criteria. The user has a few choices to decide the appearance of a graph.

The "best" layout generated according to a set of predefined criteria may not be a satisfactory one for the user since the predefined criteria cannot cover all the layout requirements in real applications. Instead, the user has important reasons to control automatic layouts tightly:

- Layout preference. The definition of nice-looking differs among users. A layout that looks good for one user may be unsatisfactory for another one. Because general layout criteria cannot take personal preference into account, users need to control automatic layouts to express special concerns.

- Semantics. Semantics of a graph must be expressed by constraints or visual organizations of a layout. The user or applications need to control automatic layouts in order to generate layouts which are not only nice-looking but also convey the correct information to viewers.

- Layout quality. Because many layout requirements lead to intractable problems, heuristics are commonly used to trade off quality and performance. A heuristic-based algorithm cannot guarantee an optimal result. In the experiment reported in [34], the best automatic layouts have edge-crossings that can be easily removed by humans. This indicates that user's instructions can improve layout quality and guide a heuristic-based algorithm to converge smoothly.

- Interactive style. The quality of interfaces directly determines the usefulness of a software system. A key to high quality interfaces is supporting direct manipulation [52]. Direct manipulation requires a system provide rapid, reversible and incremental actions. Complex language syntax is replaced by direct operations on the object of interest, e.g., by selecting and dragging with a mouse instead of using names and commands with a keyboard. This enables the user to see
immediately the effects of current activities and control the directions of the activities to achieve the goals. The user experiences less anxiety because the actions are easily reversible. The user also gains confidence because she feels that she is in control of the system.

Direct manipulation means tight control on automatic layouts. The user should be able to see the layout activities in order to diagnose layout problems. The user should also be able to control the layout activities to get the expected results.

To summarize, a layout requirement may be specific, depending on personal preference or individual situations. A layout requirement may also be unknown until a graph is seen (see Figure 1.7). The specific and dynamic requirements must be expressed through the user’s control on the layouts. Otherwise, the user either has to accept the automatic layout as “the” best layout, or manually customize it. This reduces the efficiency of the automatic method and degrades layout quality.

![Figure 1.7: Layout requirements](image)

This dissertation addresses the problem of controlling automatic layouts. The objective is to support the user to create customized layouts with several widely used layout algorithms. The approach of this dissertation can be summarized as follows:

- Increase the expressive power of existing layout algorithms.
- Introduce new layout techniques to generate structured layouts.
- Integrate different layout techniques to deal with various layout requirements from the user and applications.
- Improve the interfaces of existing layout tools to help the user diagnose layout problems and interact with the layout algorithms directly.
The result of this work is a layout tool called LYCA. LYCA uses incremental optimization algorithms to improve aesthetics of graphs. A novel usage of divide-and-conquer layout is introduced to generate structured layouts. A constraint solver and the layout algorithms are integrated to deal with layout constraints. The interfaces of LYCA are designed to support the principles of direct manipulation. With those features, users can generate customized layouts easily and flexibly.

1.3 Overview of the Dissertation

The reminder of this dissertation is organized as follows. Chapter two gives a literature review on related techniques, including layout algorithms, layout systems, and constraint-based geometry layout. Chapter three gives an overview of LYCA's architecture.

Chapter four presents layout algorithms used by LYCA. Three algorithms are described in the chapter: an improved force-directed placement algorithm, a novel divide-and-conquer layout algorithm, and LYCA's implementation of a Sugiyama style algorithm.

Chapter five presents the constraint solver of LYCA. The first half of the chapter explains the constraint solving algorithm. The latter half of the chapter explains integration of the layout algorithms and the constraint solver.

Chapter six presents the interfaces of LYCA. Several issues are covered in the chapter, interfaces for diagnosing layout problems, interfaces for controlling the layout process, parameterizing layout algorithms and some other features of LYCA's interfaces.

Chapter seven evaluates the performance of LYCA and compares LYCA with other approaches by experiments and complicated examples.

Finally, chapter eight summarizes the results of the dissertation and suggests directions for further work.
Chapter 2

Related Work

There is a growing interest on the layout problem as graphic user interfaces become widely used in all kinds of software applications. A variety of layout techniques have been proposed in the past decades. The comprehensive bibliography of [3] contains over two hundred papers on layout algorithms and layout systems. A number of areas in existing works are related to the customization problem in graph layout:

- Incremental layout
- Layout algorithms
- Constraint-based geometry layout
- Layout systems

This chapter reviews important works in the above areas and compares the advantages and disadvantages of different approaches.

2.1 Layout Modes

Most layout methods work in static mode. A static layout method takes a graph as input and outputs a layout of the entire graph in a single step. In a typical scenario, the user opens a graph file, issues a layout command, and edits the resulting layout. The last two steps may be repeated several times until a satisfactory layout is found.

In static mode, the layout process is closed to the user: the user cannot see the layout process, neither can the user influence the layout process. The entire graph has to be redrawn if the user wants to correct a problem in a layout. This may drastically alter the layout and introduce undesirable changes. The user has
weak control on layouts in static mode because the user has to manipulate layouts indirectly.

One important and difficult challenge in graph layout is to develop dynamic layout algorithms. In dynamic layout, a graph is constructed interactively. Each time the user adds a vertex or an edge to a graph, the graph is redrawn so that the layout of the new graph continues to satisfy the aesthetic criteria.

It is more difficult to perform dynamic layout because layouts must be updated very efficiently for interactive graph editing. Furthermore, the change between subsequent layouts should be minimized for "smooth" evolution of a layout. This is crucial to keep the user's mental map on an existing layout [36]. Otherwise, the user may get confused and experience difficulty to understand the new layout.

Several dynamic layout algorithms have been proposed in recent years [3]. Among the most influential works is the dynamic layout work frame and algorithms proposed by Cohen, et al [9].

A dynamic layout process is incremental and reversible. The user also can intervene in the layout process by editing the intermediate layout results. But dynamic layout may not preserve the topology of a graph. The user's modification on intermediate results may be changed by a dynamic layout algorithm as the layout evolves.

The third layout mode is the incremental mode [35]. Like dynamic layout, the user creates graphs interactively in incremental layout. But more restrictions are imposed on static layout methods. As the user edits a graph, only the newly defined vertices or edges can be positioned by static layout methods in order to preserve the topology of the existing layout.

The incremental algorithm of [35] creates layouts semi-automatically. The user positions vertices of a graph manually. The algorithm automatically decides the optimal routes to connect a new vertex to its neighbors. The incremental layout algorithm reported in [38] can shift blocks of a graph to make space for newly inserted vertices.

The user has a tight control on the resulting layouts in incremental mode because the user directly draws graphs. The drawback is that more manual work is needed in incremental layout, which can be time-consuming.
2.2 Layout Algorithms

The current layout implementation approaches can be divided into three categories: algorithmic approach, declarative approach and integrated approaches [30]. Each approach strives to achieve different goals using different techniques.

2.2.1 Algorithmic Approach

The algorithmic approach emphasizes the correctness and efficiency of a layout method. The objective is to develop algorithms which provably achieve a set of well-defined criteria with minimal computational resources. Typically the criteria are aesthetics. Most layout methods surveyed in [3] belong to this approach, e.g., algorithms for planarity testing or minimizing the number of edge-bendings, etc.

Some algorithmic methods also provide limited constraint-solving capability. The algorithm of GIOTTO [54] handles aesthetics such as crossings, bends, edge lengths and area. It also supports several constraints, including a center vertex, external vertices, shape, stream. Another example is DAG [14] and its successor DOT [40]. DAG and DOT use Sugiyama style algorithms to draw directed graphs. Users can constrain the range of a vertex along Y-Dimension in a hierarchy layout, e.g., maximal and minimal levels.

Two algorithms which belong to this approach are explained in detail: the force-directed placement and Sugiyama algorithm. Both algorithms are successful in producing aesthetically pleasing layouts. The two algorithms are adopted by LYCA to draw undirected graphs and directed graphs, respectively.

Force-directed placement

*Force-directed placement* [12] is a variation of the *spring embedding* algorithm [11]. The algorithm draws a graph by applying an analogy from a natural process. Informally, vertices of a graph are represented as atomic particles that exert forces upon each other. Like the electronic forces in the macro-cosmic world, the magnitude of the forces between vertices is related to the distance between them. However, the forces need not resemble the realistic forces in the natural world faithfully. Instead,
they are designed to reflect the layout criteria. In the model of [12], all vertices repel each other with repulsive forces. Neighbor vertices attract each other with attractive forces.

Initially, vertices are located randomly into the plane. The simulation starts by releasing the vertices. Like in the natural world, forces induce the movement of vertices. Each vertex attempts to move toward a stable location where the net force on it is zero. The concept of temperature in simulated annealing [17] is incorporated to control the motion of vertices. Vertices with high temperatures move violently to their stable locations, while vertices with low temperatures are finetuned at their positions. As the simulation goes on, vertices are cooled down. The simulation continues until all vertices are stabilized or a user-defined threshold is reached. The final state represents the generated layout of the graph.

Force-directed placement algorithm is successful in producing nice-looking layouts. The algorithm is often used by graph tools to draw undirected graphs in straight-line style. One example layout of the force-directed placement is shown in Figure 2.1. The layout is created by the graph editor GraphEd [20].

Several variations of the force-directed placement algorithm have been proposed by other researchers. The algorithm of [55] positions vertices in a certain order. A depth-first search is performed to find the center of a graph. The algorithm then adds vertices to the layout from the center of the graph. The algorithm of [13] uses
local temperatures to avoid oscillations and rotations of vertices. Both approaches improve the layout quality and the efficiency of the original algorithm.

**Sugiyama Style Algorithm**

A Sugiyama algorithm [51] draws directed graphs as hierarchy graphs. The algorithm consists of three steps:

- **Preprocessing.** This step assigns vertices to levels by a topological sort. Loops are eliminated by temporarily reversing the directions of some edges on the loops. Dummy vertices are introduced to break long edges that cross several levels.

- **Barycentric ordering.** This step rearranges the vertices on each level to reduce the number of edge-crossings between adjacent levels. The up-barycenter, down-barycenter and median barycenter represent the average positions of the immediate predecessors, immediate successors and all neighbors of a vertex, respectively. A downward pass, an upward pass and several up-down passes are performed to reduce the number of edge-crossings.

- **Fine-tuning.** The last step calculates the final x, y coordinates for each vertex.

A desired feature for the first step is to reduce the total length of edges in a hierarchy graph. The height and width of a graph should be balanced. This gives compact and nice-looking layouts. One important improvement on this step is supporting constraints along the Y-Dimension [40].

Since minimizing the number of edge-crossings is NP-hard, a heuristic is used to perform the second step. Although an optimal solution cannot be guaranteed, the algorithm generally works well in reducing the number of edge-crossings. The goal of the last step is to distribute vertices evenly, make uniform edge-lengths and straighten long paths. The algorithm of [48] uses forces to perform the fine-tune step, which gives good results.

A Sugiyama style algorithm is widely used by layout tools to draw directed graphs. One example layout of a directed graph is given in Figure 2.2. The layout is created by the graph editor DAG [14].
2.2.2 Declarative Approach

The declarative approach emphasizes the express power of a layout method. Layout requirements are usually represented as a set of constraints or a cost function. This enables the user or applications to specify different layout requirements uniformly. A graph is laid out by searching for a solution of the constraints or finding a configuration with minimal cost.

Simulated Annealing

One representative algorithm in this category is simulated annealing [17]. Simulated annealing accepts a cost function as the layout requirements. Different requirements can be specified as weighted terms in the cost function. The algorithm draws a graph by searching for a configuration with minimal cost.

In the natural world, a liquid will reach a totally ordered form, the crystal form, when the liquid is cooled slowly. A crystal form represents the minimum energy state of the liquid system. The simulated annealing algorithm draws a graph by simulating the physical process in which a liquid is slowly cool down into crystal status. The initial configuration can be chosen randomly [17] or by pre-processing algorithms [18]. The current configuration is changed randomly. If a change reduces the energy of the physical system, it is accepted. Otherwise, the change is rejected or accepted based on the Boltzmann distribution. Temperature is incorporated into the
control of the layout process. When temperature is high, molecules that represent vertices of a graph have more freedom to move toward less optimal status. As the temperature decreases, only improvement is accepted.

Genetic Algorithms

A genetic layout algorithm also accepts a cost function as layout requirements. Such an algorithm draws a graph by simulating the natural evolution process in which the most-fit survives. In [26], a genetic algorithm is used to produce visually organized layouts, as shown in Figure 2.3.

Generalized Spring Algorithm

A generalized spring algorithm is used in [10] to produce visually organized layouts. A cost function $LC = \sum \alpha_i c_i$ is formed from constraints, where $\alpha_i$ is the weight of the i-th constraint and $c_i$ is a function that evaluates the degree to which the i-th constraint is satisfied. Assume $L = < x_1, y_1, \ldots, x_n, y_n >$ is a layout, with $x_i$ and $y_i$ as the x and y coordinates of the i-th vertex, respectively. The algorithm attempts to find a small change $\Delta$ to $L$, with $\Delta = < dx_1, dy_1, \ldots, dx_n, dy_n >$, such
that \( LC(L + \Delta) < LC(L) \). Repeatedly applying the algorithm to \( L \) drives \( LC(L) \) to zero.

The algorithm features a strategic scheduling mechanism: it solves difficult constraints before solving easy ones. This increases the chance of success. A certain number of layouts are generated and the best one is chosen as the result.

### 2.2.3 Integrated Approach

The integrated approach uses an algorithmic component to improve aesthetics of a graph and a declarative component to process constraints. Because different layout requirements are processed by specialized techniques, more expressive capability can be achieved with a reasonable time performance.

**EDGE**

The graph editor EDGE [42] represents an important approach of integrating different techniques to handle the combined layout problem. In EDGE, the layout algorithms, the user, and the stability algorithm all generate constraints with priorities. The constraint solver of EDGE takes constraints from different sources. The solver first resolves conflicts among constraints. The solver then evaluates the consistent constraints and returns a solution of the constraints as the layout, as shown in Figure 2.4. The graph editor generator GEDL [22] adopts the same approach as EDGE to process aesthetics and layout constraints.

![Diagram](image)

Figure 2.4: The integrated approach of EDGE
TreeSnake

TreeSnake [30] is a graph tool that supports user-defined constraints using options provided by layout algorithms. Many layout algorithms are nondeterministic in the sense that there are multiple choices for the internal operations of those layout algorithms. Depending on the choices of those operations, a layout algorithm may return different layouts that satisfy aesthetic criteria for the same graph. For instance, using a Sugiyama style algorithm, both layouts in Figure 2.5 may be produced (from [30]). Such nondeterministic behavior provides options to help control a layout. In the case that an option of a layout algorithm has several choices and one of the choices makes the resulting layout satisfy constraints, the layout algorithm can satisfy constraints by selecting the right choice for the option. A customizable method is an integrated approach where a declarative method sets the options of an algorithmic method according to the conformance criteria or preference criteria.

TreeSnake draws binary trees with a dynamic programming algorithm. The customizable method can draw a tree in different conventions and solve constraints such as orders of vertices or sizes of subtrees.

![Figure 2.5: Nondeterminism in an algorithmic method](image)

2.3 Constraint-Based Geometry Layout

One important application of constraint programming is geometry layout. In constraint-based geometry layout, the user draws objects on a screen and attaches constraints to the objects. The behavior of an object is dominated by the constraints. As the user drags an object around on the screen, other objects are adjusted automatically to satisfy the constraints. Examples of such systems include CoDraw [15], Juno [39], etc.

Other constraint-based drawing systems take a draft drawing from the user and
Constraint programming, or logic programming, is an important research field that has drawn increased interest recently. The main concern of a constraint-based system is developing expressive and efficient constraint solving algorithms.

2.4 Other Approaches

The Diagram Sever [2] organizes algorithmic components in an object-oriented inheritance network. The components in the sever can be combined in a module fashion, e.g., as an execution path in the network. The sever accepts a set of user-defined layout criteria and automatically decides the execution path that can achieve the criteria.

Some layout methods use rules as graph grammars or layout algorithms [8, 49]. They either extract layout rules from stereotype layouts or allow the user to define layout specifications with textual rules or visual rules. A rule-based algorithm is presented in [26] to generate visually organized layouts (see Figure 2.6).
2.5 Comparison of Different Approaches

The above approaches have different advantages and disadvantages. The algorithmic approach is very successful in dealing with the readability and efficiency requirements. But the algorithmic approach overlooks the customization problem. Most algorithmic methods work in static mode. The layout requirements are predefined and hard-coded in the implementation. Only a limited constraint-solving capability is provided. It is usually difficult for the user to generate customized layouts with algorithmic layout methods.

Simulated annealing, genetic algorithms and a generalized spring algorithm are very expressive because they can accept any layout requirement as a weighted term in a cost function. But they are computationally inefficient. It can take a long time for such algorithms to find a result. To make it worse, the quality of the result may not be guaranteed. This limits their applicability in actual applications.

Constraint-based geometry layout is well suited for applications like simulation and graphic user interface design, where constraints are the main concerns. The immediate visual feedback and direct manipulation on graphic objects enable the user to control a layout easily. However, it is difficult to use constraints to represent aesthetics such as edge-crossings or symmetries.

The integrated approach of EDGE and GEDL takes the advantages of both the algorithmic and declarative approaches. It is expressive, extendible, and reasonably efficient. The problem is that the solver does not consider aesthetics when solving constraints. Consequently, the solver may break constraints derived from the layout algorithm and yield layouts that violate aesthetic criteria.

The customizable method of TreeSnake is based on an algorithmic method. It performs limited adjustments to the algorithmic method. When the options of a layout algorithm provides enough freedom to accommodate constraints, the customizable method gives an optimal solution in the sense that both aesthetics and constraints are achieved. A difficulty is to develop such methods to support various types of constraints. It is likely that sometimes aesthetics have to be violated to satisfy constraints.

Parametric or synthesized layout algorithms accept user-defined layout specifi-
cations on a class of graphs. They provide controllability to the user at a coarse granularity. But they do not deal with specific or dynamic requirements at a fine granularity.
Chapter 3
Overview of LYCA

The focus of this thesis is supporting the user to generate customized layouts easily and flexibly. To achieve this goal, a layout tool called LYCA was developed. LYCA uses different techniques to process various layout requirements. All the techniques are integrated as a comprehensive solution to the customization problem in graph layout.

This chapter presents an overview of LYCA’s architecture. Major components of LYCA and their functionalities are described briefly. The internal and external representation of graphs in LYCA are also explained.

3.1 Architecture of LYCA

The top level architecture of LYCA is shown in Figure 3.1. There are four components: Display component, Graph class, Constraint solver and Layout component. Each component processes certain kinds of layout requirements.

The layout component focuses on the aesthetic issue. It provides three layout

![Figure 3.1: Configuration of LYCA](image-url)
algorithms to draw graphs. A force-directed placement is used to draw undirected graphs in straight-line style. A divide-and-conquer algorithm is used to generate structured layouts. A Sugiyama style algorithm is used to draw directed graphs in hierarchical style.

The constraint solver processes user-defined layout constraints. This task can be divided into two sub-tasks: constraint validation and constraint evaluation. When a constraint is defined, the constraint solver validates the newly added constraint by detecting and eliminating conflicts among constraints. For the remaining constraints that are consistent, the constraint solver uses a propagation style algorithm to solve constraints. A solution of the constraints is returned as an intermediate result or the final layout.

The constraint solver is designed to work efficiently. It only maintains a small set of constraints defined by the user. In each layout iteration, the constraint solver takes a layout from the layout component and changes the positions of constrained vertices to satisfy constraints. This minimizes the overhead of integration. However, if the constraint solver and the layout component do not cooperate in some way, the solver may violate aesthetic criteria when solving constraints. To avoid such problems, the constraint solver and the layout component cooperate with each other to detect and remove congestion caused by constraints. The cooperation between the solver and the layout component improves the layout quality with a reasonable cost.

The display manager manages the display window of LYCA. A basic functionality of a display manager is to show layouts on the screen. But LYCA’s display manager goes beyond that. As noted before, it is difficult to control automatic layouts if the layout process is invisible. LYCA’s display manager visualizes the layout process at different levels of abstraction. Visualization makes the layout process understandable and predictable. The user can easily diagnose problems in a layout and decide how to correct the problems.

LYCA’s display manager also supports the user to control automatic layouts in more active ways: it allows the user to interact with the layout algorithms in a dynamic context. The user can pause, rewind, forward or resume the layout process as a layout is being generated. Manual layout can be applied to the intermediate results of the layout process. This allows the user to control automatic layouts more
flexibly. It also improves the interactive style between the user and the layout tool because the user can see the layout activities and control the layout activities directly.

3.2 Data Structure and Graph Representation

3.2.1 Internal Representation

Graphs are represented in LYCA as instances of the Graph class. LYCA uses a hygraph style structure to represent graphs: each vertex has its own lists of edges and vertices. The top level vertex is the graph, and each vertex in the graph can represent a subgraph. Therefore, nested graphs can be represented with LYCA’s data structure.

The Graph class also provides methods that manipulate a graph. Some important methods are:

- LoadGraph: load a graph and its layout data.
- SaveGraph: save a graph and its layout data.
- Close: clean vertices and edges in a graph.
- AddVertex: add a vertex to a graph.
- AddEdge: add an edge to a graph.
- DeleteVertex: delete a vertex.
- DeleteEdge: delete an edge.
- EdgeNo: return the number of edges.
- VertexNo: return the number of vertices.
- GetEdges: find an edge by id or attribute.
- GetVertices: find a vertex by id or attribute.
- GetParent: return parent graph.

Attributes of graphs are stored in attribute lists, which are instances of the AttributeList class. The operations on AttributeList include:

- AddAttribute: add an attribute.
- DeleteAttribute: delete an attribute.
- ChangeAttribute: change the value of an attribute.
The Edge class defines data structure and functions for edges. Notice that there is no Vertex class since Graph class and Vertex class are used interchangeably. More on LYCA's design can be found in appendix A.

### 3.2.2 External Representation

Externally a graph is represented in MERA format. MERA (Meta Entity Relation Attribute) is a versatile graphic language developed by Software Engineering Research Laboratory (SERL), University of Hawaii [53]. A feature of MERA is that it can be used to define different graphic languages by creating a meta-graph as the formalism of the new language. Currently MERA has the capacity for animation, view operation, user adaptability, and some semantic level process. The language is used in SERL for modeling and analysis of user requirements, building prototypes, capturing designer's knowledge and controlling the design process.

MERA language is the standard format for knowledge representation and knowledge exchange in the software maintenance environment developed by SERL. Hence LYCA can be integrated with other tools developed in SERL to serve as their user interfaces.

A graph in MERA format typically has two parts: definitional data and the visualization data. The definitional part contains specifications on formalism, attribute
name/value pairs, object id, connections between objects, etc. The visualization part contains information on locations of vertices, icon, display attributes, etc. MERA allows the two parts to be stored in different files so multiple views can be created for a given graph. Following is an example MERA file that defines the graph shown in Figure 3.2.

```
D-DIAGRAM("DM2","test", [])
D-ENTITY(20, "test", "data_item", "RD", [])
D-ENTITY(30, "test", "record", "item1", [])
D-ENTITY(40, "test", "record", "item2", [])
D-ENTITY(60, "test", "space", "DS", [])
D-RELATION(100, "test", "contain", "redef-1", 30, 20, [])
D-RELATION(120, "test", "contain", "redef-2", 40, 20, [])
D-RELATION(140, "test", "belong", "belong-1", 20, 60, [])

V-VIEW("Default", ",", "test", ",", [])
V-ENTITY("Default", 20, 874, 947, 0, [V-SHOW-ATTRIBUTE(F)])
V-ENTITY("Default", 30, 749, 1034, 0, [V-SHOW-ATTRIBUTE(F)])
V-ENTITY("Default", 40, 996, 1038, 0, [V-SHOW-ATTRIBUTE(F)])
V-ENTITY("Default", 60, 874, 838, 0, [V-SHOW-ATTRIBUTE(F)])
V-RELATION("Default", 100, 0, [V-SHOW-ATTRIBUTE(F)])
V-RELATION("Default", 120, 0, [V-SHOW-NAME(T)])
V-RELATION("Default", 140, 0, [V-SHOW-ATTRIBUTE(F)])
```

In the example, formalism and default view information are defined by the metagraph named as “DM2”. The first part of the file defines entities and relations of a graph. The second part of the file gives locations and display attributes of objects in the graph.

LYCA also uses its own file format to represent layout data such as graph style, layout parameters and user-defined constraints. Those will be discussed separately in following chapters.
Chapter 4
Layout Algorithms

The layout component of LYCA handles requirements on aesthetics and structure of a graph. It provides three layout algorithms to draw graphs. A force-directed placement is employed to draw undirected graphs. A Sugiyama style algorithm is employed to draw directed graphs. A divide-and-conquer approach is introduced to generate structured layouts. This chapter presents the three layout algorithms.

4.1 Undirected Graphs

LYCA uses force-directed placement [12] to draw undirected graphs. A force-directed placement algorithm draws a graph by applying an analogy from the electronic forces in the macro-cosmic world. In the model, vertices of a graph are represented as atomic particles that exert forces upon each other. All vertices repel each other with repulsive forces. Neighbor vertices of edges attract each other with attractive forces. The strengths of forces are defined as:

\[ F_r = -k^2/d \]
\[ F_a = d^2/k \]

where \( F_a \) is the attractive force, \( F_r \) is the repel force, \( d \) is the distance between a pair of vertices, \( k \) is a constant defined by the user. Notice that the repulsive force and attractive force between a pair of vertices cancel each other when the distance between the vertices is equal to \( k \). Therefore, \( k \) represents the radius of the desired empty area around each vertex.

Force-directed placement algorithm is quite successful in improving several aesthetic criteria of a graph, including even distribution of vertices, uniform edge-lengths, a small number of edge-crossings, inherent symmetry and conformance to the frame. The algorithm is often used by layout tools to draw undirected graphs.
Figure 4.1: Layouts created by force-directed placement

The force-directed placement algorithm in [12] assumes that all vertices in a graph have the same size. If this assumption does not hold, the algorithm scales up the entire layout by the size of the largest vertex to avoid overlaps between vertices. As illustrated by the example in Figure 4.1, if a graph contains vertices of the same size, the algorithm of [12] returns the layout shown in Figure 4.1 (a). If the user increases the radius of one vertex in the graph, the algorithm returns the layout shown in Figure 4.1 (b). The second layout is unpleasantly large because vertices are sparsely distributed in the layout and part of the graph becomes invisible. The layout is also not balanced since each vertex occupies the same area no matter what is the actual size of the vertex.

Most graphs in real applications contain vertices of different sizes. The force-directed placement in [12] may return poor layouts for graphs with large vertices. It is hence useful to improve the algorithm to generate nice-looking layouts for graphs that contain vertices of different sizes.
Figure 4.2: Distance between large vertices

Assume \( v \) and \( w \) are a pair of vertices with areas \( S_v \) and \( S_w \), respectively. If \( d \) is the straight line that connects the centers of \( v \) and \( w \), \( d \) is divided into three fragments: the portion of \( d \) inside \( S_v \) is denoted as \( d_{in}^{v} \), the portion of \( d \) inside \( S_w \) is denoted as \( d_{in}^{w} \), the portion of \( d \) between \( S_w \) and \( S_v \) is denoted as \( d_{out} \), as shown in Figure 4.2.

The algorithm in [12] returns large layouts for graphs with vertices of different sizes because it does not consider \( d_{in} \) when calculating forces. To overcome this problem, the force formulas of [12] are modified as follows:

\[
F_a = \begin{cases} 
0 & \text{if } w, v \text{ overlap} \\
\frac{d_{out}^2}{k'^2 + d_{in}^2} & \text{otherwise}
\end{cases}
\]

\[
F_r = \begin{cases} 
C \frac{k'^2}{d} & \text{if } w, v \text{ overlap} \\
\frac{k'^2}{d} & \text{otherwise}
\end{cases}
\]

where \( d \) is the distance between the centers of a pair of vertices, \( d_{in} \) is the portion of \( d \) inside the areas of the vertices, e.g., \( d_{in}^{w,v} = d_{in}^{w} + d_{in}^{v} \), \( d_{out} \) is the portion of \( d \) between areas of the vertices, \( C \) is a constant decided by experiment.

The revised algorithm has the following two features:

- When a pair of vertices overlap, the attractive force between them is zero. The repulsive force is increased to repel the two vertices away from each other.

- When the distance between the areas of a pair of vertices is equal to \( k' \), i.e., \( d_{out} = k' \), the repulsive force and attractive force between the vertices cancel each other.
The improved algorithm differs from the original algorithm in adjusting distances between vertices with areas. The original algorithm ignores shapes and dimensions of a pair of vertices when calculating the optimal distance between them. In the improved algorithm, two neighbor vertices reach a stable status when the distance between their areas is $k'$. Therefore, $k'$ represents the radius of the desired empty area around a vertex's area.

Figure 4.3: Compact layout generated by LYCA (40 percent of actual size)

The improved algorithm has several advantages when drawing graphs with unequally sized vertices. It ensures that layouts are syntactically valid by avoiding overlaps between vertices. The layouts produced by the improved algorithm are more compact than those produced by the original algorithm. In addition, vertices are evenly distributed if we consider sizes of vertices in the distribution. An example layout generated by the improved algorithm is given Figure 4.3. The example shown in Figure 4.4 has more vertices with different sizes.

Another application of the improved force-directed placement algorithm is to draw meta graphs in divide-and-conquer layout. This usage is explained in the next section.
4.2 Divide-and-conquer Layout

4.2.1 Basic Concepts

The following concepts are used in the discussion of divide-and-conquer layout.

Given a graph $G = (V, E)$, a partition, $P$ splits $G$ into disjoint subgraphs: $P = \{G_1, \ldots, G_n\}$ such that

$$G_i = (V_i, E_i)$$

$$\bigcup_{i=1}^{n} V_i = V$$

$$V_i \cap V_j = \emptyset \text{ for } i \neq j$$

$$E_i = \{(v, w) \in E | v, w \in V_i\}$$

$P$ also divides the edges of $G$ into two categories: intra-edges and inter-edges. An intra-edge is an edge with both endpoints in the same subgraph.

$$E_{\text{intra}} = \bigcup_{i=1}^{n} E_i$$

An inter-edge is an edge with the two endpoints in different subgraphs.

$$E_{\text{inter}} = E - E_{\text{intra}}$$
Given a graph $G$ and a partition $P = \{G_1, \ldots, G_n\}$ of $G$, an undirected graph $G_{\text{meta}}$ called a \textit{meta-graph} is constructed by collapsing subgraphs of $G$ into \textit{meta-vertices} and transforming inter-edges of $G$ into \textit{meta-edges}:

\[
G_{\text{meta}} = \{V_{\text{meta}}, E_{\text{meta}}\} \\
V_{\text{meta}} = \{G_1, \ldots, G_n\} \\
E_{\text{meta}} = \{(G_i, G_j)|w \in G_i, v \in G_j, (w, v) \in E_{\text{inter}}\}.
\]

A layout of $G_{\text{meta}}$ is called a \textit{meta-layout} of $G$. A meta-layout can be obtained from a layout of $G$ by setting the dimensions and center of each meta-vertex as the dimensions and center of the underlying subgraph, respectively. In reverse, giving a meta-layout and layouts of subgraphs, a layout of $G$ can be obtained by replacing the meta-vertices and meta-edges with the underlying subgraph layouts and actual inter-edges, respectively.

\[
\begin{array}{c}
\text{Figure 4.5: Meta graph and meta layout} \\
\end{array}
\]

In the example shown in Figure 4.5, graph $G$ is partitioned into three subgraphs. Given the layout of $G$ on the left side, the corresponding meta-layout is shown on the right side. In reverse, given the meta-layout on the right side and the layout of the subgraphs, a layout of $G$ can be obtained.

In the rest of the thesis, the terms \textit{structured layout} and \textit{organized layout} are used interchangeably.

\subsection{Generating Structured Layouts}

As noted before, a visually organized layout is one in which vertices are grouped by proximity, symmetries, zones, shapes, and other constraints derived from the
Gestalt Laws of perceptual psychology. The Gestalt Laws represent our innate rules of organizing scenes in the seeing process. A layout that follows those rules helps the user understand the graph. A layout that violates those rules may mislead the user.

LYCA adopts a divide-and-conquer approach for generating structured layouts. Conceptually, a divide-and-conquer approach draws a graph in three steps:

- partition a graph into subgraphs.
- draw subgraphs independently.
- compose subgraph layouts to form the resulting layout.

Divide-and-conquer layout can be used to generate structured layouts since it is relatively easier to deal with the complex constraints on the structure of a graph when subgraphs are laid out in isolation. For example, if vertices are placed closely in subgraphs and subgraphs are placed apart from each other, proximity of vertices in the same subgraph can be reflected in the final layout.

A divide-and-conquer approach is also useful to display zones and symmetries of subgraphs. Although the force-directed placement of [12] can display inherent symmetry of a graph, it does not work well if only part of a graph is symmetric. This is because the unbalanced forces between the symmetric part and the unsymmetric part of a graph may twist the shape of the former. In divide-and-conquer layout, forces between subgraphs are masked in subgraph layout so that symmetries of symmetric subgraphs can be displayed precisely in the resulting layout.

Divide-and-conquer layout, on the other hand, has the problem that inter-edges are totally ignored in subgraph layouts. This may cause long edges and/or edge-crossings in a resulting layout. The problem is illustrated by the example shown in Figure 4.6. The layout in Figure 4.6 is generated by a simple divide-and-conquer algorithm which does not consider inter-edges in subgraph layouts. The layout is not nice-looking because inter-edges are improperly positioned in the layout.

There is no trivial solution to this problem since inter-edges depend on subgraphs which in turn depend on inter-edges. This leads to a circular dependency problem. For example, if we want to calculate the attractive force between \( n_{13} \) and \( n_{14} \) when drawing subgraph \( S_1 = \{n_3, n_2, n_{14}\} \), we need to know the distance between \( n_{14} \)
and n13 as a vector with direction and value. This means that we need to know the layout of subgraph $S_2 = \{n9, n10, n11, n12, n13, n18\}$ and the locations of $S_1$ and $S_2$ in the resulting layout. However, the layout of $S_2$ also depends on the layout of $S_1$ for the same reason.

Previously proposed divide-and-conquer approaches either require manual modifications of a resulting layout or recursive adjustment of subgraph layouts [19, 40]. In LYCA, a novel usage of the divide-and-conquer approach is introduced to generate structured layouts.

Given a partition, forces in the force-directed placement algorithm are classified into inter-forces and intra-forces. A force between a pair of vertices in the same subgraph is called an intra-force. A force between a pair of vertices in different subgraphs is called an inter-force. Let $P = \{G_1, \cdots, G_n\}$ be a partition of a graph $G = (V, E)$, where $G_i = (V_i, E_i)$. For vertex $v \in G_i$, the intra-force on $v$ is

$$F_{\text{intra}}^v = \sum_{w \in V_i} F_{r, w}^v + \sum_{(v, w) \in E_i} F_{a}^{(v, w)}$$
The inter-force on $v$ is:

$$F_{\text{inter}}^v = \sum_{w \in V, j \neq i} F_{r}^{v,w} + \sum_{(v,w) \in E_{\text{inter}}} F_{a}^{(v,w)}$$

A composite force on a vertex $v$ is defined as:

$$F_{\text{comp}}^v = F_{\text{intra}}^v + S(t)F_{\text{inter}}^v + (1 - S(t))F_{\text{meta}}^v$$

where $F_{\text{comp}}^v$ is the composite force on $v$, $F_{\text{intra}}^v$ is the intra-force, and $F_{\text{inter}}^v$ is the inter-force. $F_{\text{meta}}^v$ is the force derived from the meta-layout corresponding to the current layout. As mentioned above, given a partition of graph $G$ and a layout of $G$, a meta-layout of $G$ can be constructed by collapsing subgraphs into meta-vertices and transforming inter-edges as meta-edges. For each meta-layout, the improved force-directed placement algorithm is used to calculate forces between meta-vertices in the meta-graph. The net force on a meta-vertex is the meta-force on all vertices contained by the subgraph that is represented by the meta-vertex. For example, in Figure 4.5, the meta-forces on $A$, $B$ and $C$ in subgraph $S1$ are equal to the net force on meta vertex $S1$ in the meta-layout on the right side.

![Figure 4.7: Scheduling function S](image)

$S(t) \in [0, 1]$ is a function of layout time $t$ such that $S(t)$ decreases as $t$ increases after a threshold $t'$ and reaches 0 at another threshold $t'' (> t')$, as shown in Figure 4.7.

The divide-and-conquer algorithm uses composite forces to position vertices. Initially, $t = 0$, $S(t) = 1$, and $F_{\text{comp}} = F_{\text{inter}} + F_{\text{intra}}$. Between time 0 and time $t'$, the original force-directed placement algorithm [12] is used to create a layout with uniform edges and a small number of edge-crossings. But the layout may not be well structured because subgraphs may be overlapped and symmetries may not be reflected properly. Figure 4.8 (a) shows an example layout at time $t'$.  

35
As the layout process proceeds, $S(t)$ decreases. This reduces the influences of inter-edges on vertices. Meanwhile, meta-forces are increased. The meta-forces make vertices in each subgraph move like one rigid object. At the time threshold $t''$, $S(t)$ becomes zero, $F_{\text{comp}} = F_{\text{intra}} + F_{\text{meta}}$. Since vertices in the same subgraph have equal meta-forces, meta forces do not influence the relative positions of vertices in a subgraph. This means that vertices in subgraphs are positioned independently just like in a divide-and-conquer layout. The layout generated at time $t'$ is restructured to reflect symmetries and shapes of subgraphs. The meta-forces resolve overlaps between subgraphs and display zones of subgraphs. This yields a organized layout that displays proximity, symmetry, zones and shapes of subgraphs. Furthermore, because the final layout is gracefully evolved from the layout generated at time $t'$ which satisfies aesthetic criteria, the final layout also inherits good aesthetics. Therefore, the resulting layout is well-structured as well as aesthetically pleasant,
as shown in Figure 4.8 (b). Figure 4.10 shows the final layout of another example generated by the divide-and-conquer layout.

![Figure 4.9: Subgraphs in divide-and-conquer layout](image)

The layout process of the divide-and-conquer algorithm can be regarded as a transformation process. Initially, boundaries of subgraphs are soft such that external forces can penetrate the boundary and act upon vertices inside subgraphs. As the layout process proceeds, boundaries of subgraphs harden so external forces are weaken when penetrating the boundaries. At time \( t'' \), boundaries of subgraphs become solid and completely block external forces from outside, as shown in Figure 4.9.

![Figure 4.10: Another example of divide-and-conquer layout](image)
4.2.3 Weights of Meta-Edges

One subtle problem in the divide-and-conquer layout is how to decide weights of meta-edges and meta-vertices in the meta-graph. The force-directed placement algorithm in [12] accepts weighted edges and vertices. If a pair of vertices are connected by an edge with weight $w$, the strength of the attractive force between them is:

$$F_a = w \frac{d^2}{k}$$

In meta-graph, a meta-edge may represent multiple inter-edges. If the weight of each meta-edge is set as the sum of the weights of the underlying inter-edges, attractive forces may become too strong so that meta-vertices are placed very closely in the meta-layout.

Ignoring weights of meta-edges also has problems. Since the number of inter-edges between two subgraphs is not reflected in the meta-edge, the attractive force between a pair of meta-vertices that represent subgraphs with multiple inter-edges may be too weak to overcome resistance forces from other meta-vertices. This may cause long edges in the final layout.

LYCA takes a compromise solution. Assume meta edge $e_m$ represents a set of inter-edges $E_{e_m} = \{e_1, e_2, \ldots, e_n\}$, the weight of $e_m$ is set as:

$$w(e_m) = \max(w(e_1), w(e_2), \ldots, w(e_n)) + \frac{n - 1}{C}$$

where $C$ is a constant. For example, if $C$ is equal to 10 and all inter-edges have weight 1, the weight of a meta edge representing three inter-edges is equal to 1.2. The weight of a meta edge representing one inter-edge is equal to 1. This avoids meta-edges with big weights while somehow reflecting the connections between subgraphs.

Assume a meta vertex $v_m$ represents subgraph $S = \{V, E\}$, the weight of $v_m$ is set as:

$$w(v_m) = \max(v|v \in V)$$
4.3 Directed Graphs

LYCA uses a Sugiyama style algorithm [51] to draw directed graphs. As described in chapter two, a Sugiyama style algorithm draws a graph in three steps:

- Preprocessing: assign vertices to levels.
- Barycentric ordering: rearrange orders of vertices on each level to reduce the number of edge-crossings.
- Fine-tune: calculate \( x, y \) coordinates for each vertex.

4.3.1 LYCA's Implementation of Sugiyama Algorithm

Because the original algorithm in [51] does not consider constraints, the three steps of the algorithm are modified in LYCA's implementation to deal with user-defined layout constraints and cooperate with the constraint solver.

Preprocessing

This step assigns levels to vertices by a topological sort. Cycles in a graph are temporarily eliminated by reversing the directions of some edges in the graph. If a path crosses several levels, dummy vertices are introduced at all intermediate levels to ensure that each edge crosses only one level.

In LYCA's implementation, the user can define absolute constraints along the Y-Dimension. This enables the user to assign vertices to levels manually.

Barycentric Ordering

The goal of the second step is to reduce edge-crossings between adjacent levels. Since this problem is known as NP-hard, a heuristic is used in [51]. The heuristic, called barycentric ordering, locates each vertex close to its barycenter. A barycenter of a vertex is the average position of its neighbors. A down-barycenter of a vertex is the average position of its predecessors. An up-barycenter of a vertex is the average position of its successors. A median-barycenter is the average position of both successors.
and predecessors. The algorithm performs several upward, downward, and up-down passes through the graph until no more improvement can be made or a user-defined threshold is reached.

The algorithm in [51] uses matrices to represent orders of vertices in each level. But the algorithm does not assign x coordinates for vertices in this step. It is difficult to solve constraints with only ordering information. For example, we cannot solve the constraint “$A.x = B.x$” by assigning $A$ and $B$ the same order number in their levels because we can shift all the vertices in a level toward one direction. This changes the x coordinates of the vertices but does not change their order in the level.

One way to solve the problem is to convert orders of vertices into constraints as in the graph editor EDGE. But this requires that the constraint solver update a large number of constraints in each layout iteration. LYCA takes a different approach. It uses a force-version of the barycentric ordering heuristic to sort vertices and assign x coordinates to vertices. The pseudo code of LYCA’s implementation of the second step is given below:

One-pass-reordering: direction
1) for vertex v in graph g
   set force on v as distance to barycenter
endfor
2) move vertices by forces
3) sort vertices in each level by x coordinates
4) solve constraints
5) adjust vertices in each level

The algorithm first uses x coordinates of vertices to perform barycentric sorting (step 1-3). It then invokes the solver to solves constraints (step 4). This may break orders assigned by the barycentric sorting. In step 5, the algorithm attempts to preserve the barycentric orders of vertices by adjusting free vertices in each level. The pseudo code for this step is given below:

Adjust-vertices-in-one-level: level
1) for v=leftmost vertex; v; v = right neighbor of v
2) if v is not constrained
In the step, the solver attempts to insert free vertices into the gaps between constrained vertices in order to preserve the orders of vertices in a level. If there is enough space to accommodate vertices the orders of which are broken, they are squeezed into the space to preserve their order in the level. Otherwise, the algorithm places vertices close to their barycenters but does not preserve the orders of the vertices calculated by the barycentric ordering heuristic.

The above heuristic works reasonably well when a small number of constraints are defined on vertices in each level. But in certain cases, shifting and oscillation can occur, which suggests further improvement.

**Finetune**

The Sugiyama algorithm does not explain how to perform the fine-tune step. Therefore different implementations usually use their own algorithms for this step.

LYCA uses an algorithm based on the force-directed placement approach to perform the fine-tune task. The objective is to distribute vertices evenly, make edge-lengths more uniform, and straighten long-paths. Because we just need to calculate the x coordinate for each vertex, forces along the Y-Dimension are not needed.

In the fine-tune step, the attractive force on a vertex \( v \) is set as the distance between the current position of \( v \) and the optimal position of \( v \):

\[
F_a = opt\_position(v) - v.x
\]

The optimal position is calculated by the algorithm according to the barycenter and edges of a vertex. If two neighbor vertices are too close, they repel each other. The strength of the repel force between neighbors \( v \) and \( w \) is:

\[
F_r = opt\_gap(v, w) - \text{abs}(v.x - w.x)
\]
The optimal gap between two neighbors is calculated from the widths of the two vertices and a user-defined constant.

Under certain situations, a large block of vertices in a level can be congested in the fine-tune step. To avoid such problem, the fine-tune algorithms detects congestion in each level. If several vertices are blocking each other, they are temporarily treated as a single rigid object and are moved toward the direction of the weighted average of forces on them. The pseudo code for the fine-tune algorithm is:

1) for each vertex v in graph g
2) set Fr on v as opt_pos(v) - v.x
   end-for
3) for each pair of neighbors v, w in a level
4) if abs(v.x - w.x) < opt_dist(v,w)
5) set Fa as opt_dist(v,w) - abs(v.x - w.x)
   end-if
   end-for
6) for each level
7) if some vertices are congested and have no space to move
8) set the force on each vertex as the weighted average of the forces on the congested vertices
   end-if
   end-for
9) calculate new x for each vertex
10) solve constraints

Figure 4.11 shows a layout of a directed graph from an example in [14].

4.3.2 Straighten Long Paths

One desired function in the last step is to straighten long paths consisting of dummy vertices. Otherwise, long paths may be displayed as bent lines in the final layout. LYCA uses preferential force to straighten long paths. A preferential force attempts to solve a constraint which is desired but not required in a layout. Such a constraint is called a preferential constraint. For example, the user may require that a critical
Figure 4.11: Another layout of a directed graph

path in a PERT diagram be laid out as a straight line and prefers to straighten simple paths in the PERT diagram. In the fine-tune step, preference forces are introduced into the force-directed placement algorithm to push dummy vertices in a long path toward the weighted center of the path, as shown in Figure 4.12.

Figure 4.13 shows a layout of a directed graph from an example in [47] and redrawn by LYCA. In the layout, long paths are straightened by preferential force.

LYCA also allows the user to straighten long paths manually. The user first drags dummy vertices to the center of the long path. LYCA enforces the user’s editing by further straightening the long paths which are roughly straightened by the user when it is difficult for the user to straighten the entire path. In the example shown
in Figure 4.14, several long paths are too close and one long path is not straight. The user then manually straighten long paths, as shown in Figure 4.15. LYCA detects the long paths roughly straightened by the user and enforces the user's editing, resulting the final layout in Figure 4.16.
Figure 4.14: Initial layout
Figure 4.15: After manual editing
Figure 4.16: Straightening long paths interactively
Chapter 5
Constraint Solver

A layout constraint is a spatial relation in the drawing of a graph. Layout constraints are the most important means to control automatic layouts. With constraints, users or applications can express layout preference, semantics, and other special requirements in automatic layouts explicitly and consistently.

This chapter presents LYCA's constraint solver. The first part of the chapter explains how the solver represents and validates layout constraints. The second part of the chapter presents the constraint solving algorithm adopted by LYCA. The third part of the chapter describes how to integrate the solver and the layout component in order to improve layout quality. The last part of the chapter gives examples of layouts with constraints.

5.1 Overview

As noted before, there are three basic approaches that have been proposed to support constraints in graph layout. The algorithmic approach suggests a single comprehensive algorithm that is capable of handling both aesthetic criteria and layout constraints. The declarative approach uses a general optimization algorithm or constraint solver to generate layouts. The integrated approach integrates an algorithmic component with a declarative component to process aesthetic criteria and layout constraints separately.

LYCA takes an integrated approach to process layout constraints. Like EDGE and GEDL, LYCA uses a constraint solver to process constraints. But LYCA's approach is different from the approach of EDGE in several aspects.

In EDGE, layout algorithms produce constraints from aesthetic criteria and stability criteria. The solver takes constraints from different sources and resolves con-
flicts among constraints. The solver then returns a solution of the consistent con-
straints as a layout. In spite of its advantages, the integrated approach of EDGE
has two drawbacks. The first problem is the efficiency of EDGE. EDGE's layout
component and solver communicate with constraints. The solver must validate and
evaluate a large number of constraints in each layout iteration. Because constraint
validation is a costly operation, validating many constraints repeatedly can increase
the layout time significantly.

Another problem of EDGE is that the solver and layout component do not coop-
erate. If conflicts arise between constraints produced from aesthetics and constraints
defined by the user, EDGE's solver breaks algorithm-generated constraints to sat-
ify user-defined constraints. But the solver does not inform the layout algorithms
which constraints are broken to satisfy user-defined constraints; neither do the solver
and the layout algorithm communicate to avoid breaking constraints produced from
aesthetic criteria. Therefore, EDGE may return layouts which satisfy user-defined
constraints but violate aesthetic criteria.

The two problems are addressed in LYCA. The constraint solver of LYCA does
not communicate with the layout component by constraints. Instead, the solver only
keeps constraints defined by the user. Once a constraint is validated, it will not be
checked until new constraints are added. In each iteration, the solver inputs a layout
from the layout component and changes positions of constrained vertices to satisfy
user-defined constraints. The solver and the layout component also cooperate to
detect and resolve congestions caused by solving constraints.

The integrated approach of LYCA has several advantages. Because LYCA’s solver
does not need to validate a large number of constraints repeatedly, it works more
efficiently than the solver of EDGE. In addition, the solver and the layout component
cooporate to improve layout quality, as explained in this chapter.

The architecture of LYCA’s constraint component is shown in Figure 5.1. There
are two kind of solvers in LYCA. The high level solver manages low level solvers. It
also provides the following functions for other components of LYCA:

- Define, delete, and change constraints
- Query status of constraints and vertices
Save and load constraints

Solve constraints

Cooperation method with the layout component

The low level solvers validate and evaluate constraints. There are two low level solvers because LYCA adopts the same assumptions as EDGE:

- Constraints along different dimensions are independent.
- Constraints are linear equations.

The two assumptions ensure efficient evaluation of constraints and reduce the difficulty of implementation. A wide range of constraints can be expressed under the assumptions. Therefore they do not impose severe restrictions on layout applications in practice.

In accordance with the assumptions, the two low level solvers manage constraints along the X-Dimension and the Y-Dimension, respectively.

5.2 Constraints

LYCA's constraint solver can process three kinds of layout constraints:

- *Absolute constraints* constrain a vertex's position in regard to a fixed coordinate system. For example, constraint "A.y F X" fixes vertex A at its current location in the Y-Dimension.
• Relative constraints constrain a vertex’s position in relation to other vertices. For example, “A.y > B.y” and “A.x = B.x + 24.”

• Clustering constraints group several vertices and edges into a subgraph, which can be laid out independently or constrained as a whole.

A formal definition of the constraint language of LYCA is given in appendix A. In the following, the usage of each kind of constraints is explained with examples.

5.2.1 Absolute Constraints

An absolute constraint fixes a vertex in one or two dimensions. The format of an absolute constraint is:

\[
\text{name[.dim] FX [: P=priority]}
\]

The vertex to be constrained is identified by its name. The direction field “dim” defines the dimension along which the vertex should be fixed. The value of the “dim” field can be “x” or “y.” If this field is omitted, the high level solver generates two low level constraints to fix the vertex along both the X-Dimension and the Y-Dimension. “FX” is the keyword which specifies the type of the constraint. Priority is used in resolving conflicts among constraints. When several constraints conflict with each other, the solver removes constraints with lower priority to maintain the consistency of constraints.

The constraint “A.x FX” fixes vertex A along X-Dimension. The constraint “B FX : P = 5” fixes vertex B in both dimensions with priority 5.

5.2.2 Relative Constraints

The most commonly used constraints are relative constraints. A relative constraint specifies a spatial relation between a pair of vertices. As mentioned before, each constraint must be a linear equation. The following relative constraints can be defined:

• EQ or =. An “equal” constraint means that two vertices should be placed at the same point in the dimension.
• GT or >. A “greater than” constraint means that one vertex should be placed after or below another vertex.

• LT or <. A “less than” constraint means that one vertex should be placed before or above another vertex.

• CT. A “center” constraint means that one vertex should be placed in the center of a set of vertices.

• NB. A “neighbor” constraint means that two vertices must be placed as adjacent neighbors in a level. This constraint only applies to directed graphs.

Each relative constraint takes the format:

```
name1[dim] rel name2[dim] adjust [: P=priority] [: R=method]
```

The fields “dim” and “priority” are the same as in absolute constraints. The “rel” field specifies the type of a relative constraint. The field “adjust” is an expression which specifies an adjustment value. In the current implementation, “adjust” can be an positive or negative integer. For example, the constraint “A.x > B.x + 24” means that the x coordinate of vertex A must be greater than the x coordinate of vertex B plus 24. Constraint “A = C + 24” aligns vertex A to vertex C with the margin of 24 in both dimensions.

The “R” option is used to define the reference vertex in a constraint. Many constraints can be solved in several different ways. For example, constraint “A = B” can be satisfied in three ways: change the position of A to the position of B, change the position of B to the position of A, or set both A and B to a new position. If the position of a vertex is changed to satisfy a constraint, the vertex is called a constrained vertex. If the position of a vertex is used to set the positions of other vertices in a constraint, the vertex is called a reference vertex. By default, the solver chooses the second vertex in a relative constraint as the reference vertex. In another word, the solver will change the position of the first vertex to satisfy a relative constraint. To change this default setting, the user can use the “R” option to specify the reference vertex.

For example, LYCA’s solver will solve constraint “A.x < B.x” by changing the position of A. However, if another constraint, “A.x = C.x,” is already defined, the
position of \( A \) cannot be set twice to solve both constraints, e.g., once \( A \) is set to be equal to \( C \), it cannot be changed unless \( C \) is also changed. LYCA solves this problem by using rigid constraint validation rules. The advantage is that the solver does not need to propagate values of each variable in two directions when solving constraints. This makes the solver more efficient. The drawback is that the user must choose methods for constraints carefully. In the above example, if the second constraint is given as \( A.x < B.x : R = B \) and LYCA finds choosing \( B \) as the reference vertex conflicts with the first constraint, it will try to change the reference vertex to \( A \), which does not contradict the first constraint.

5.2.3 Cluster Constraints

A cluster constraint groups several vertices into a subgraph. The format of cluster constraint is

\[
\text{Cluster( name-list ) : [SN=subgraph-name]}
\]

The keyword “Cluster” defines the type of the constraint. Field “name-list” gives names of the vertices to be grouped. In the current implementation, subgraphs must be disjoint and cannot be nested. The constraint “Cluster( A B C D E )” defines a subgraph that contains five vertices.

The “SN” option allows the user to assign a string as the name of a subgraph. This option is useful if the user wants to define constraints on subgraphs.

In LYCA, cluster constraints are mainly used to generate structured layouts with the divide-and-conquer layout algorithm. This constraint can be used only for undirected graphs.

5.2.4 Weights of Edges

It is sometimes useful to assign several weights to an edge in the divide-and-conquer layout. A weight constraint changes the weight of an edge at a time threshold in the divide-and-conquer layout process. This feature is useful to mask undesirable forces. For example, in Figure 5.2 (a), the attractive forces of edges \((b5, b2)\) and \((b6, b2)\) are
masked to display subgraph $S = \{b_1, b_2, b_3, b_4, b_5, b_6\}$ as a regular circle. Otherwise, $S$ will be displayed as a twisted shape, as in Figure 5.2 (b).
5.2.5 Constraint File

Constraints are stored in constraint files. Each constraint file is a regular text file which begins with the keyword "Constraint-Begin" and ends with the keyword "Constraint-End." The body of a constraint file is a list of constraints in text format. One example of a constraint file is given below. The file defines the constraints used to create the layout in Figure 5.2 (a).

Constraint-Begin
6.x = 1.x
3.y = 5.y
a4.x = a3.x - 60
a3.x = a2.x - 60
a1.x = a5.x - 70
c2.x = c3.x - 70
c3.x = c1.x - 70
c5.x = c4.x - 60
a5.y = a1.y
a4.y = a3.y
a3.y = a2.y
a2.y = a1.y + 64
c3.y = c2.y
c2.y = c1.y
c4.y = c5.y
c5.y = c1.y + 64
c6.y = c5.y + 64
b2-b5.w = 0
b6-b2.w = 0
Constraint-End

Constraint files can be created with the constraint editor provided by LYCA. The user also can create a constraint file with any text editor. A constraint file can be loaded by the user from LYCA. If the constraint file is included in the layout data file with the same name as the graph file, LYCA will load the constraints automatically when the corresponding graph file is loaded. The suffix of a layout file is "lyca."
The suffix of a MERA graph file is “.cd.” For example, for a MERA graph file with name “match-flight.cdf,” the default layout file has name “match-flight.lyca.”

When a constraint file is loaded, the high level solver parses each constraint and decides which low level solver is responsible for the constraint. If there are syntax errors in a constraint definition, the solver displays warning message on the screen and abandons the constraint.

### 5.3 Constraint Graphs

The low level solvers use *constraint graphs* to represent constraints. A constraint graph is a special graph that defines a set of constraints with special edges and vertices in it. The types of edges and vertices that form a constraint graph are:

- **Graph**
- **Vertex**
  - ConstraintVertex
    - N-Vertex
    - C-Vertex
    - O-Vertex
- **Edge**
  - ConstraintEdge
    - FX-edge
    - EQ-edge
    - GT-edge
    - LT-edge
    - CT-edge
    - NB-edge
    - Value-edge

An “N-vertex” defines a pointer to a vertex in the user graph. A “C-vertex” defines a constant or expression. An “O-vertex” represents an arithmetic operator. Edges in a constraint graph represent constraints. The direction of a constraint edge defines the method to solve the constraint: the solver propagates values from sources to
targets. Therefore, the source vertex is the reference vertex in a constraint since its position is used to set the position of the corresponding target vertex.

An example of constraint graph is given in Figure 5.3. The constraint graph defines the Y-Dimension constraints that are used to create the layout in Figure 5.23. In this constraint graph, only EQ-edges and N-vertices are used.

Since a constraint graph is an instance of the Graph class, all methods defined by the Graph class can be applied to a constraint graph. In addition, the low level solvers provide specialized methods to process constraints defined by a constraint graph, including:

- Constraint query. Check whether a constraint can be derived from the user-defined constraints.
- Status query. Check the status of a constraint or all constraints related to a vertex or a subgraph.
- Define constraint. Check whether a constraint conflicts with existing constraints. If there is no conflict, the constraint is added to the constraint graph.
Otherwise, all constraints involved in the conflict are returned to the high level solver.

• Delete constraint. Remove a constraint from the constraint graph.
• Solve constraints. Change positions of N-vertices in the constraint graph to satisfy constraints.
• Cooperation with the layout component. This function is explained in the section on integration.

5.3.1 Constraint Query

A constraint query checks whether a given constraint can be derived from a constraint graph. For example, given constraints “A > B” and “B = C,” the constraint “A > C” can be derived.

The low level solver processes a constraint query by checking whether there is a special path in the constraint graph. This can be done with a breath-first search. The constraints that can be queried are:

• FX. Vertex u is fixed iff u is fixed by a “FX” edge or is equal to a vertex that is fixed.
• EQ. Vertex u is equal to vertex v iff u and v are connected by an “EQ” edge or u is equal to a vertex that is equal to v.
• GT. Vertex u is greater than vertex v iff
  - there is an “GT” edge from u to v, or
  - there is an “LT” edge from v to u, or
  - u is equal to or greater than a vertex that is greater than or equal to v, or
  - the leftmost position of u is greater than the rightmost position of v.
• LT. Vertex u is less than vertex v iff
  - there is an “LT” edge from u to v, or
- there is an "GT" edge from $v$ to $u$, or
- $u$ is equal to or less than a vertex that is less than or equal to $v$, or
- the rightmost position of $u$ is less than the leftmost position of $v$.

• Related. Vertex $u$ is related to vertex $v$ if there is a path from $u$ to $v$. All edges in the path must be traversed in the same direction.

Constraint queries are important to maintain the consistency of constraints because to validate a new constraint, we need to find out whether the new constraint conflicts with any constraint that can be derived from existing constraints.

5.3.2 Consistency

LYCA's constraint solver keeps two set of constraints. The high level solver keeps a list of constraints in text format. The constraint list can be browsed and edited by the user. At the low level, each low level solver keeps a constraint graph to represent constraints in one dimension. When the user defines a new constraint, the high level solver invokes the low level solver to update constraint graph. The pseudo-code for this procedure is

Define_Constraint
  if no conflict
    add constraint to constraint graph
  else
    while conflict && a low priority constraint is involved
      remove the low priority constraint
    endwhile
    if no conflict
      add constraint to constraint graph
    endif
  endif

To validate a new constraint, the low level solver checks whether a constraint that contradicts the new constraint can be derived from its constraint graph. For example, to validate constraint "$A.x < B.x$", the solver checks whether any of the following
queries return true: \( A.x > B.x \), \( A.x = B.x \), \( A.x \neq B.x \). The solver also checks whether the leftmost position of \( A \) conflicts with rightmost position of \( B \). If all the queries return false, the new constraint is consistent with existing constraints and is accepted. Otherwise, the constraints that are involved in the contradiction are identified for resolving conflicting.

The following rules are adopted by LYCA for constraint validation:

- **FX.** Vertex \( u \) can be fixed if it is not a constrained vertex in any constraints.
- **GT.** Vertex \( u \) can be constrained to be greater than \( v \) if \( u \) is not fixed and \( v \) is not greater than or equal to \( u \).
- **LT.** Vertex \( u \) can be constrained to be less than \( v \) if \( u \) is not fixed and \( v \) is not less than or equal to \( u \).
- **EQ.** Vertex \( u \) can be constrained to be equal to \( v \) if \( u \) is not fixed and there are no direct or indirect relations between \( u \) and \( v \).
- **CT.** Vertex \( u \) can be constrained as the center of a set of vertices if it is not a constrained vertex in any constraints and it is not related with a fixed vertex.
- **NB.** Vertex \( u \) can be constrained as the neighbor of vertex \( v \) if \( u \) is not fixed and \( v \) is not equal to \( u \) and there is no vertex which is greater than \( v \) and less than \( u \).

The above rules assume that vertex \( u \) is chosen as the constrained vertex.

If a contradiction is detected, the low level solver reports error to the high level solver. All constraints involved in the contradiction are also sent back to the high level solver. The high level solver attempts to resolve the conflict by removing the constraint with the lowest priority or choosing another method for the newly defined constraint.

For example, assume all the constraints in Figure 5.3 have the default priority 3. If the user tries to define a new constraint \( c4.y > c6.y : P = 5 \), the new constraint conflicts with the old constraints because the constraint \( c4 = c6 + C \) can be derived from the existing constraints, as shown in Figure 5.4. The constraints involved in the contradiction are shown as dashed lines. Because the new constraint has higher
priority, the high level solver removes existing constraint, "\(c6 = c5 + 64\)," to resolve the conflict. The new constraint is now consistent with the existing constraints and is added to the constraint graph, as shown in Figure 5.5.

The next example shows a contradiction that is different from the first example. In the constraint graph shown in Figure 5.6, vertices \(n1\) and \(n2\) are fixed at position 300 and 260, respectively. Vertex \(n3\) is constrained to be less than \(n2\). Notice that although there is no path between \(n1\) and \(n3\), the relation "\(n3 < n1\)" can be derived from the positions of \(n1\), \(n2\) and the constraint "\(n3 < n2\)." If the user tries to define the constraint "\(n3 > n1\)," it conflicts with the existing constraints since the relation "\(n3 < n1\)" can be derived from the constraint graph, as in Figure 5.7. If all constraints have equal priority, the solver will reject the new constraint.

### 5.4 Solving Constraints

A consistent constraint graph is a directed, acyclic graph. The root vertices in the graph are "C-vertices" or "N-vertices" that are used only as reference vertices in constraints.
Figure 5.5: Consistent constraint graph

Figure 5.6: Constraint graph

Figure 5.7: Conflicting constraints
Constraints are evaluated by a propagation style algorithm which propagates values with a depth-first traversal progressing from roots to leaves of the constraint graph. The algorithm first assigns a leftmost position and a rightmost position for each “N-vertex.” It then searches the constraint graph to find a vertex that is ready to be solved. A vertex is ready to be solved if it is a root or all input constraints to the vertex are set. If such a vertex is found, the position of the vertex is calculated and propagated to all the vertices that are constrained by the vertex. This makes the positions of more vertices solved. The process repeats until all vertices are solved.

Following is the pseudo code of the evaluation process:

```
Evaluate_Constraint_Graph(constraint graph G)
    while find vertex v in G that is ready to be solved
        set position of v
        propagate(v)
    endwhile

Propagate(vertex v)
    for each outgoing edge of v
        if the target vertex of the edge is ready to be solved
            set position of the target vertex
            propagate(the target vertex)
        endif
    endfor
```

Since LYCA uses rigid rules to validate constraints, the solver can evaluate a constraint graph by propagating values in one direction. The propagation algorithm takes $O(n)$ time to solve a constraint graph.

### 5.5 Examples of Solving Constraints

Figure 5.8 shows an example of a constraint graph. This constraint graph is quite simple. To solve the constraints, the algorithm first finds that $n1$ is ready to be set since it is a root in the constraint graph. The position of $n1$ is assigned as the position calculated by the layout algorithm. The value of $n1$ is used to set the value
of n2, which is used in turn to set the position of n3. The constraint graph is solved in one iteration.

![Diagram](image)

**Figure 5.8:** Example of evaluating constraint graph

The second constraint graph (see Figure 5.8) contains more constraints. The algorithm first sets the position of n1 as the position given by the layout algorithm. The value of n1 is used to set n2. The value of n2 is then propagated to n3. Because n3 is constrained by both n2 and n4 but only n2 is set at this point, the propagation stops at n2. The algorithm returns to n1 and propagates the value of n1 to n3. The last step is to set the position of n4 according to the positions of n2 and n3.

![Diagram](image)

**Figure 5.9:** Another example of evaluating constraint graph

The last example (see Figure 5.10) uses leftmost and rightmost positions. The constraint graph is shown in Figure 5.10. In the graph, two vertices, n1 and n2, are fixed at positions 200 and 400, respectively. Vertex n5 is constrained to be greater than n1 and less than n2. Therefore, the range of n5 is between 201 and 399. n5 is also constrained to be greater than n4. It implies that the rightmost position of n4 is 398. Otherwise, constraint “n5 > n4” cannot be satisfied.
Given the above information, the solver sets the position of $n1$ and $n2$ in the first two iterations. In the third iteration, the solver tries to set the position of $n4$. It first checks whether the current position assigned by the layout algorithm is within the range of $n4$. If this is true, the position assigned by the layout algorithm is accepted as the position of $n4$. Otherwise, the solver chooses a position that is close to the algorithm-assigned position from the range of $n4$. The value of $n4$ is propagated to $n5$ and $n3$. The constraint graph is evaluated in three iterations.

5.6 Integration

As mentioned in the review chapter, two approaches have been proposed previously to integrate constraint solvers and layout algorithms. The approach of EDGE [42] integrates solver and layout algorithm by constraints. The approach of TreeSnake generates customized methods based on the options provided by layout algorithms.

EDGE requires that the layout algorithm produce constraints from aesthetics. The solver accepts constraints from different sources, resolves conflicts, and returns a solution of the constraints as the layout of a graph. This approach has the advantage that all layout requirements are uniformly represented as constraints. This means that EDGE can process any layout requirements as long as the requirements can be converted into constraints.
A problem of EDGE is the one-way communication between the solver and other layout algorithms. When algorithm-generated constraints contradict user-defined constraints, EDGE's solver breaks algorithm-generated constraints to resolve the conflict. But the solver does not inform the layout algorithm which constraints generated by the algorithm are broken in resolving conflicts, neither does the solver cooperate with the layout algorithm to avoid breaking constraints from the algorithm. This may result in layouts that satisfy user-defined constraints but violates aesthetic criteria.

TreeSnake [30] represents another integrated approach. It uses options provided by layout algorithms to satisfy constraints. If a layout algorithm can provide adequate options to deal with a kind of constraint, TreeSnake can return optimal layouts that achieve both aesthetics and constraints. A problem is that if aesthetics have to be violated to satisfy constraints, TreeSnake cannot deal with such kinds of constraints. The types of constraints that can be supported by this approach depend on the flexibility of the layout algorithm.

In LYCA, the solver inputs a layout from the layout component and changes positions of constrained vertices to satisfy user-defined constraints in each iteration. This simple integrated approach is very efficient. But it suffers from the problem that the solver solves user-defined constraints "brutally" without taking care of aesthetic criteria. This can jeopardize the convergence of the layout process and cause poor layouts. As illustrated in the following example, the graph in Figure 5.11 (from [17])
Figure 5.12: Interaction between solver and layout algorithm

is drawn with the force-directed placement algorithm with two constraints, “n7.x = n8.x + C” and “n6.x = n17.x - C.” This layout has two problems: 1) It occupies a large area; 2) There is a long edge in the layout. Analyzing the layout process step by step reveals why solving constraints results in the poor layout. In Figure 5.12, the problematic part of the graph in Figure 5.11 is separated from the rest of the graph to simplify the analysis. The animation in Figure 5.12 shows the interaction between the layout algorithm and solver in one layout iteration. At the beginning of the layout iteration, the net force on each vertex is calculated by the layout algorithm (see Figure 5.12 (a)). The layout algorithm moves vertices according to the net forces on them, as shown in Figure 5.12 (b). At this point, n7 and n17 are moving toward the left while n8 and n6 are moving toward the right. However, the new configuration calculated by the layout algorithm violates the constraints defined by the user. The solver then changes the positions of n7 and n6 to satisfy constraints, as shown in Figure 5.12 (c). This means that the attractive force between n7 and n6 is completely masked because of solving the constraints. As the layout process continues, the solver keeps moving n7 and n6 to follow the movement of n8 and n17, respectively, causing the poor layout in Figure 5.11.
The example indicates that solving constraints can make a vertex block other vertices from reaching their optimal positions assigned by the layout algorithm. If constraints cause a vertex to block other vertices from being optimized, the vertex is referred to as a c-barrier. The occurrence of c-barriers indicate that improving aesthetics has reached the threshold of constraints. To further improve the aesthetics of the graph, both constraints and aesthetics must be taken into account.

Figure 5.13 shows several examples of c-barriers. In Figure 5.13, the movement for each vertex is shown as the arrow on the vertex. Because the movements proposed by the layout algorithm violate the constraints, they will be suppressed or denied by the solver.

As mentioned before, the force-directed placement algorithm is a variation of the spring embedding algorithm [11]. In the model, vertices are connected by springs which exert attractive forces and repulsive forces upon the vertices. If we do not change the force formulas, it does not matter whether electronic forces or spring forces are used to position vertices. So we adopt the model of the spring embedding algorithm in the following discussion.

To represent constraints in the simulation model, we introduce “steel sticks” into the model. Each steel stick has a length and is attached to a vertex. Unlike a spring, a steel stick does not produce forces. Instead, it can propagate forces from one end to the other end. If two vertices reach the threshold of a constraint and one vertex is blocking the other from reaching its optimal position, the two ends of the steel-stick that represents the constraint touch the two vertices to propagate forces between them, as shown in Figure 5.14. The force on the blocked vertex is propagated to the blocking vertex to push it away. The force on the blocking vertex is also propagated back to the blocking vertex to prevent it from breaking constraint. This makes the two vertices behave like a single rigid object until the blocking disappears.
For example, the constraint \( A < B - 24 \) can be represented as a steel stick of length 24. If we attach the steel stick to \( A \), the other end of the steel stick will not touch \( B \) when \( A \) is less than \( B - 24 \). When \( A \) has reached the threshold of the constraint, i.e., \( A = B - 24 \), and \( B \) becomes a c-barrier to \( A \), the steel stick connects \( A \) and \( B \) and propagates forces between them. This means that the solver not only changes the position of \( A \) to solve the constraint, it also informs the layout algorithm when \( B \) is blocking \( A \) and how \( B \) should be adjusted to achieve the best overall aesthetics of both \( A \) and \( B \) while keeping the constraint. The corresponding pseudo code is given below:

\[
\text{Force-Directed-Placement( Graph: G )}
\]

1) While( not done )
2) calculate forces
3) solver->adjust_forces
4) calculate position
5) solver->solve_constraints

end while

In the pseudo code, the layout component invokes solver twice (step 3 and step 5). In step 3, the layout algorithm and the solver cooperate to process c-barriers by propagating forces between blocked and blocking vertices. The pseudo code for this function is:

\[
\text{adjust-forces( layout-forces, vertices )}
\]

1) for all constraints
2) switch( type of constraint )
3) center constraint:
4) distribute the net force on the centered
5) equal constraint:
6) neighbor constraint:
7) calculate the average force on a vertex
8) set forces on all vertices as the average force
9) greater than constraint:
10) less than constraint:
11) if c-barrier
12) propagate forces
end if
end switch
end for

In the algorithm, if several vertices are aligned by equal or neighbor constraints, they are always treated as a single rigid object. The force on each vertex is the weighted average of all the forces on the vertices that are aligned by the constraints. If a vertex is constrained to be at the center of a set of vertices, the force on it is evenly distributed on the constraining vertices. If one vertex is blocking another vertex because of a relative constraint, the blocking and blocked vertices are temporarily treated as a single rigid object. Notice that a c-barrier itself also can be blocked by other vertices. Therefore, the function actually detects groups of blocking and blocked vertices and treat all the vertices in such a group as one object.

The cooperation between layout component and solver effectively reduces c-barriers and improves layout quality, as shown in Figure 5.15. Compared with the layout in Figure 5.11, the layout in Figure 5.15 satisfies both constraints and aesthetic criteria.

C-barriers also can occur for directed graphs. For example, the layout in Figure 5.16 has two edge-crossings. If constraints "$D NB D3$" and "$E NB E1$" are defined on the graph, the layout generated by a solver and a layout component that do not cooperate with each other has 11 edge-crossings, as shown in Figure 5.17.

The problem happens because the Sugiyama algorithm tries to place each vertex close to its barycenter. However, constraints make some vertices block others from
Figure 5.15: Layout improved by the cooperation between the solver and layout algorithm

Figure 5.16: Original layout

reaching their barycenters. This is why LYCA uses a force-version implementation to perform the barycentric ordering: forces are used to bias the barycenters of constrained vertices and blocking vertices to avoid poor layouts caused by constraints. When c-barriers are processed by cooperation between solver and layout algorithm, the resulting layout has 4 edge-crossings, as shown in Figure 5.18.
Figure 5.17: Layout with constraints

Figure 5.18: Improved layout of directed graph with constraints
5.7 Layout Examples

This section demonstrates the layout capability of LYCA with examples.

The first example explains how to display proximity of subgraphs with cluster constraints. LYCA allows the user to define optimal distances between subgraphs \((k')\) and optimal distances between vertices \((k)\) in subgraphs in divide-and-conquer layout. If the user requests LYCA to place subgraphs sparsely and vertices in subgraphs closely, proximity of vertices can be displayed. In the layout shown in Figure 5.19, two constraints are used:

\[
\text{Y-Dimension Constraint:} \\
n_{12}.y = n_{18}.y
\]

\[
\text{Cluster Constraint:} \\
\text{Subgraph}(n_{10} \ n_{9} \ n_{13} \ n_{12} \ n_{18})
\]

The cluster constraint groups vertices \(n_{9}, n_{10}, n_{12}, n_{13}\) and \(n_{18}\) into a subgraph. The constraint on Y-Dimension ensures that edge \((n_{12}, n_{18})\) is parallel to the horizontal line. In addition, \(k\) is set to be equal to \(k'\), which means the optimal distance between subgraphs is equal to the optimal distance between vertices in subgraphs. This yields the layout that shows the shape and zone of the subgraph.

In the layout shown in Figure 5.20, \(k'\) is set to be greater than \(k\). This causes vertices in the subgraph to be placed close to each other. The layout thus displays the cohesion of vertices in the subgraph by proximity.

In the second example, constraints are used to control the structure of a graph. The layout without constraints is given in Figure 5.21. This layout is not well organized since it does not display proximity, zones, shapes and other visual organizations defined in [26]. To create visually organized layouts, the following constraints are defined on the layout:

\[
\text{X-Dimension:} \\
a_{3}.x - a_{4}.x = a_{2}.x - a_{3}.x = 60 \\
a_{5}.x - a_{1}.x = 70 \\
c_{1}.x - c_{3}.x = c_{3}.x - c_{2}.x = 70
\]
Figure 5.19: Layout that reflects zone and shape of subgraph

Figure 5.20: Layout that reflects proximity
c4.x - c5.x = 60

Y-Dimension:

\[
\begin{align*}
a5.y &= a1.y \\
a4.y &= a3.y = a2.y = a1.y + 64 \\
c3.y &= c2.y = c1.y \\
c4.y &= c5.y = c1.y + 64 \\
c6.y &= c5.y + 64 \\
3.y &= 5.y
\end{align*}
\]

Cluster Constraints:

\[
\begin{align*}
\text{Subgraph}(1 2 3 4 5) \\
\text{Subgraph}(b1 b2 b3 b4 b5 b6) \\
\text{Subgraph}(a1 a2 a3 a4 a5) \\
\text{Subgraph}(c1 c2 c3 c4 c5 c6)
\end{align*}
\]

Weight Constraints:

\[
\begin{align*}
b2-b6.weight &= 0 \\
b2-b5.weight &= 0
\end{align*}
\]

The constraints divide the graph into four subgraphs. Two of the subgraphs are constrained as hierarchy graphs. The weight constraints mask the attractive forces of edges \((b2, b6)\) and \((b2, b5)\) in the later stage of the divide-and-conquer layout. The layout created by the divide-and-conquer algorithm at time \(t'\) is given in Figure 5.22. The final layout is given in Figure 5.23. The final layout is structured and satisfies aesthetic criteria.

In the third example, the graph in Figure 4.8 is redrawn with constraints. The subgraph that contains \(n15, n16, n17\) and \(n7\) is drawn as a “T-shape,” as shown in Figure 5.24.

In the fourth example, three layouts of the same graph are given (see Figure 5.25 and Figure 5.26). The three layouts have different visual structures and thus convey different information to users.

The next example is a layout of a directed graph. The initial layout generated
Figure 5.21: Layout without constraints

Figure 5.22: Layout created at time $t'$
Figure 5.23: Resulting layout

Figure 5.24: Example layout with constraints

77
Figure 5.25: Unstructured layout

(a) A structured layout  (b) Another structured layout

Figure 5.26: Layouts that convey different information

by the Sugiyama algorithm is given in Figure 5.27 (a). Three constraints are defined to generate the layout shown in Figure 5.27 (b): "E1.x > E2.x," "C.x = D2.x," "C3.x < C2.x."

The last example graph is taken from [16]. The manual layout used in [16] is shown in Figure 5.28 (a). The automatic layout with constraints is given in Figure 5.28 (b).
Figure 5.27: An example layout of directed graph with constraints

Figure 5.28: Another example layout with constraints
Chapter 6
Interfaces of LYCA

The quality of the interfaces of a software system directly determines the usefulness of the system. In graph layout, the interfaces of a layout tool can effect the efficiency of the layout task and the quality of the resulting layouts. One of the objectives of this work is to investigate how to help users generate customized layouts with interface techniques.

This chapter presents the interfaces of LYCA. The first half of the chapter discusses how to support the user in diagnosing layout problems and controlling the layout process. The latter half of the chapter describes the interfaces to parameterize layout algorithms and debug constraints.

6.1 Animation

Generating customized layouts means to draw graphs according to user-defined criteria instead of system-defined criteria. One way to do so is to allow the user to intervene in the automatic layout process directly. This is possible since most layout algorithms are nondeterministic in the sense that their internal operations have multiple options. In TreeSnake, the options are controlled by a customized method generated from a set of user-defined criteria. If the user can set the options of the internal operations of a layout method directly, the user will be able to control the resulting layouts tightly.

For layout algorithms that perform incremental optimization, the algorithms attempt to find a good layout by repeatedly improving the current layout into a better one. The initial configuration often influences the resulting layout significantly. Therefore, if the user can access and edit the intermediate results of an algorithm
that performs incremental optimization, the user can have more control on the final layout.

The user's involvement in the layout process can improve layout quality. Many layout problems are computationally intractable. A heuristic-based algorithm cannot guarantee an optimal layout. In such cases, the user's assistance can help the heuristic-based algorithms perform uphill-climbing optimization.

The first logical step to controlling the layout process is to visualize it. Like many layout tools, LYCA visualizes the layout process as animations by displaying snapshots of the layout process continuously. Figure 6.1 shows several snapshots of the layout process of the force-directed placement algorithm. In the animation, vertices look like objects that are moved by Newtonian mechanics forces.

Because the user can see a layout algorithm at work, the layout process becomes more understandable and predictable. Consequently, if a problematic layout is returned, the user has a better chance to find the cause of the problem and figure out how to correct the problem.

For example, localization is a common problem for algorithms that perform incremental optimization (see Figure 6.2). There is no trivial solution to this problem since uphill-climbing usually means exponential complexity. In the layout shown in Figure 6.3, vertices n25, n26 and n27 attempt to move toward the right-downward direction but are blocked by vertices n7 and n17. Such a class of problems are called barrier problems [12] because some vertices block others from reaching their optimal positions. The algorithm of [12] uses the heuristic that repulsive forces are turned off periodically (every 15th iteration) to remove barriers. The heuristic can remove simple barriers. But if multiple vertices are involved in a blocking situation, the heuristic does not work well.

With animation, the user may notice that n25, n26 and n27 are oscillating back and forth around some positions in the layout process. This is a typical symptom of the barrier problem. The user then performs manual uphill-climbing by dragging n25, n26 and n27 over n7 and n17. This yields the layout shown in Figure 6.4.

Another common problem is that a layout may be constantly shifted by the force-directed placement algorithm toward one direction until the layout touches boundaries of the frame. This may cause vertices near the boundaries to be squeezed
Figure 6.1: Animation
Figure 6.2: Inferior local minima and manual up-hill climbing

Figure 6.3: Poor layout caused by barriers

together. In Figure 6.5, the shifting and squeezing problems result in poor layout with an edge-crossing and closely placed vertices. Such problem can be easily detected from animation. The user can solve the problem by moving the entire graph toward the center of the frame and fixing one or a few vertices in the plane, as shown in Figure 6.6.

Animation can elicit interests and enjoyment from users. This is useful for training new users. Animation is also important for algorithm designers and programmers. It is a difficult task to develop layout algorithms since designers and programmers often lack a clear picture of the working of an algorithm and the interaction between competing components. This can make the designers or programmers frustrated during the development process. Animation alleviates this problem to a certain extent.
Figure 6.4: Layout without barrier

Figure 6.5: Poor layout caused by squeezing problem

For example, LYCA is a medium-sized program with over 20,000 lines of C++ code. Animation was very helpful in detecting both design flaws and implementation errors in LYCA. An even more powerful way to assist algorithm designers is to combine animations with other program visualization techniques and symbolic debuggers.
6.2 Visualization of Aesthetics

Animation provides the user with an overview of the layout process. But animation itself is not enough for the user to understand the layout process because animation shows too much information in a short period. A layout tool can draw a medium-sized graph in a few seconds. It is difficult for the user to diagnose a problem from animation before the layout process finishes.

Playing animation in slow motion does not solve the problem. Unless the animation is played very slowly, the user can not understand details of the layout process. The speed of the animation depends on the size of a graph and human’s capability of digesting information on the screen. This will increase the layout time significantly, making the performance of a layout tool unacceptable.

It is also difficult for the user to evaluate the quality of a layout objectively from animation. For example, to find out the number of edge-crossings, the user must count edge-crossings one by one, which is almost impossible when hundreds of snapshots are displayed on the screen in seconds.

To alleviate those problems, LYCA visualizes the layout process at two levels of abstraction. On the one hand is the animation of the layout process, which enables the user to find out problems such as shifting and squeezing. It also helps the user analyze the layout process in detail when animations are played in slow motion. On
the other hand, LYCA provides an abstract view of the layout process. Instead of displaying animations of the layout process, LYCA displays aesthetic measures directly. The class "LayoutMonitor" defines methods to perform this task. Each instance of the "LayoutMonitor" class manages a monitor window. The monitor window is hidden until the user chooses the "Monitor" command from the "View" menu in LYCA’s menu bar. Once a layout monitor is activated, it opens its monitor window beside the main window of LYCA, as shown in Figure 6.7.

Aesthetic measures are displayed in the analysis window at the top area of the monitor window. The X-Dimension of the analysis window represents layout time. The Y-Dimension of the analysis window represents the values of aesthetic measures. When the layout process starts, the layout monitor checks the aesthetics of intermediate result in each layout iteration and displays the analysis results as plot lines that grow from left to right in the analysis area. The user or application can register multiple methods to check different aesthetics. Each aesthetic measure is represented as a plot line with unique color. For examples, the plot in red color represents the number of edge crossings, the plot in green color represents the size of the area that a layout covers.

During the layout process, if the red plot keeps going downward, it indicates that the layout algorithm is working well in reducing the number of edge crossings. However, if the plot goes up and down like a sawtooth (see Figure 6.8), it indicates that the layout algorithm is experiencing difficulties in removing edge crossings.

Figure 6.7: Layout monitor window
Heuristics cannot guarantee monotonic improvement of the aesthetic. Therefore, a sawtooth-like or upward plot indicates that the user should intervene in the layout process to overcome problems encountered by the layout algorithm.

The advantage of displaying aesthetic measures is obvious. The plots of aesthetics indicate the working of the layout algorithm straightforwardly. The user can monitor the layout process with a glance at the monitor window. Therefore, the user can diagnose layout problems without significantly increasing the layout time.

The layout monitor can be used in combination with animations. They complement each other by providing layout information at different levels of detail. The layout monitor and animations also can be turned off by the user. This minimizes the overhead on the layout process.

Currently LYCA allows the user to check several aesthetic measures, including the area of a layout, the number of crossings, the maximal edge length, the minimal edge length, the average length of edges, the standard derivation of edge lengths, and the minimal distance between vertices.

### 6.3 Controlling the Layout Process

Once the user can see the layout process, the next step is to intervene in the layout process directly. The `LayoutMonitor` class defines methods and interfaces for the
user to control the layout process. Below the analysis window is a row of control buttons that represent the following control functions:

- Pause the layout process.
- Rewind the layout process.
- Forward the layout process.
- Resume the layout process.
- Clean the monitor window.

The control buttons enable the user to interact with the layout algorithm in a dynamic context. To support interactions between users and layout algorithms, each layout monitor maintains an internal stack which stores intermediate layout configurations during a layout process. It is likely that when the user notices an interesting event and decides to take action, the animation has already passed that point of the event. In such a case, the user can use the control buttons to suspend the layout process and rewind the layout process to a given point. As the user rewinds the animations, snapshots of the layout process are displayed in reverse order. This is like playing animation frame by frame manually. The user thus can analyze the layout process at a detailed level.

When the layout process is suspended, the user can edit the intermediate configurations. For example, the user can perform manual uphill-climbing by dragging vertices over their barriers, or the user can define layout constraints to enforce certain spatial relations in the layout. The user also can resume the layout process by pressing the resume button.

The control interfaces make the layout process incremental and reversible. Those features are very important for graph layout. It is common that several layouts have to be generated before a satisfactory one can be found. If the user finds that one trial layout is not good, the user can undo the layout action easily and go back to the initial layout. The user also can select an intermediate layout as the initial layout for the next trial. This gives the user’s flexibility to maximize desired changes while avoiding undesirable changes. Otherwise, the user either has to reject the new layout, or has to correct old problems as well as new problems in the next trial. In
the worst case, the user may totally lose control of the layout and get stuck in the layout process.

The following example shows how visualization and control interfaces are used together to control the layout process. The graph in Figure 6.10 is a planar graph. But the force-directed placement algorithm returns a layout with one edge-crossing (see Figure 6.10 (a)). This is because the force-directed placement algorithm does not remove edge-crossings directly. Instead, it attempts to do so by placing connected vertices closely. But the heuristic fails in this case. To solve the problem, the user intuitively moves vertex $n7$ to the left side to remove the edge-crossing (see Figure 6.10 (b)). When the user requests LYCA to redraw the graph, the edge-crossing comes back. It is hard to find from animation why the edge-crossing comes back because the layout is generated in less than 2 seconds. On the other hand, the problem is clearly shown in the monitor window: the red plot that represents the number of edge-crossings goes up instead of goes down (see Figure 6.9). The user thus rewinds the animation manually to check what caused the problem. In the first few frames of the animation, the user notices that shortly after the layout process is started, $n7$ was pulled by attractive forces to move toward left side and bypassed $n8$, causing the edge-crossing in the resulting layout (Figure 6.10 (c) to Figure 6.10 (e)). The user thus defines constraint "$n7.x < n8.x$" and redraws the layout from the edited layout (the layout shown in Figure 6.10 (b)). This time the layout constraint prevents $n7$ from bypassing $n8$, yielding the layout in Figure 6.10 (f).
6.4 Visualization of Constraints

When many constraints are defined, it can become difficult for the user to manage the constraints. For example, the user may attempt to move a vertex in the direction that violates the constraints defined previously. Inappropriately defined constraints may also cause problems for the layout algorithm. Even for experienced users it is sometimes confusing to figure out why her editing actions do not work as expected or why the layout tool behaves strangely.

To help the user understand and debug constraints, LYCA visualizes the status of constraints defined on each vertex. If a vertex is used in a constraint, LYCA displays a rectangular border around the icon of the vertex (see vertex n7 in Figure 6.10 (f)). Such a border is referred to as a constraint-border or a c-border in the following
discussion. The color of a c-border indicates the status of the constraints defined on
the vertex.

LYCA classifies the status of vertices which are used in constraints into three
categories: “Reference,” “Satisfied” and “Unsatisfied.” As mentioned in the chapter
on constraints, most constraints can be solved in several different ways. For a relative
constraint between a pair of vertices, the solver may solve the constraint by changing
the position of either vertex in the constraints. If the position of a vertex is used
by the solver to set the position of another vertex, the vertex is referred to as a
reference vertex. Otherwise, it is referred to as a constrained vertex. If there is only
one method to solve a constraint, all vertices in the constraint are called constrained
vertices.

During the layout process, the solver does not change the positions of reference
vertices. Whether or not the solver changes the position of a constrained vertex
depends on whether the vertex violates constraints. If the user does not understand
how the solver works, the user may have difficulty predicting or understanding the
behavior of the layout tool. For example, if the user attempts to drag a constrained
vertex to a position that violates constraints, this editing action will be denied by
the solver.

LYCA uses status to indicate whether the position of a vertex may be changed by
the solver. A vertex is defined as in “Reference” status if it is only used as reference
vertex in all constraints defined on it. A vertex is in “Satisfied” status if it is used
at least once as a constrained vertex and all constraints in which it is used as the
constrained vertex are satisfied. Otherwise, a constrained vertex is in Unsatisfied
status.

If a vertex is in Reference status, the color of its c-border is blue. If a vertex is
in Satisfied status, its c-border is in green color. If a vertex is in Unsatisfied status,
its c-border has red color. The user can redefine colors for each status with the color
pallet in the command zone in LYCA’s main window.

Status of a vertex is updated when a vertex is moved by the user or by the layout
tool. For example, if the user drags a constrained vertex around on the screen,
the c-border of the vertex will become red when the position of the vertex violates
constraints. This indicates that the editing action may not work as expected because
the solver may change the position of the vertex to solve constraints. If all c-borders have blue or green color, it indicates that constraints are satisfied in the current configuration.

C-borders can be hidden if the user chooses the Constraint command in the View menu twice.

### 6.5 Parameterizing Layout Algorithms

The layout algorithms used in LYCA can be parameterized by the user. To parameterize a layout algorithm, the user chooses the commands in the Layout menu. LYCA displays different windows for the user to set parameters of different algorithms. The window that sets the parameters of the force-directed placement algorithm is shown in Figure 6.11. The layout options that can be set by the user include: animation, grid, cooling function, cluster. The animation option turns on or turns off the animation mode. The grid option forces vertices being placed on grid points in the frame. The cooling function determines the rate that a vertex cools down. The cluster option enables or disables cluster constraints. When this option is set as on, subgraphs will be laid out independently after a time threshold, as explained in the chapter on layout algorithms. The user also can change the optimal distance between vertices, the optimal distance between subgraphs, the maximal layout iteration, and the animation speed.

The window that sets the parameters of the Sugiyama style algorithm is shown in Figure A.12. The user can change layer height, width of the gap between vertices in the same layer, the maximal layout iteration, the strength of preference forces for straightening long paths. Some options for the Sugiyama algorithms are the same as the spring algorithm and thus are skipped here.

The window that sets the parameters of the finetune algorithm is shown in Figure A.13. This algorithm is a customized version of the force-directed placement algorithm and therefore is similar to the window for the force-directed placement algorithm. But some options for the force-directed placement algorithm are disabled for the finetune algorithm to ensure the algorithm does not change a layout drastically.
Figure 6.11: Parameterize force-directed placement algorithm

6.6 Other Interfaces of LYCA

Besides the interfaces described above, LYCA also provides various interfaces for the user to control solvers, display managers, or other components. The operational manual in appendix A explains how to use LYCA in detail. This section briefly describes some interesting interfaces of LYCA that are not mentioned in the previous description.

The main window of LYCA is shown in Figure 6.14. The window is divided into three parts: display window (bottom right), command zone (bottom left) and
Figure 6.12: Parameterize Sugiyama algorithm

menu bar (top). The display window is where LYCA displays layouts and animations. Three layout commands are defined in the command zone: "Undirected", "Directed" and "Finetune." The layout commands draw a graph as an undirected graph, a hierarchy graph, or finetune the current layout, respectively. The command zone also defines commonly used editing commands. The icons of the editing commands are self-explanatory and easy to use.

There are six menus in the menu bar. The "File" menu defines commands that operate on an entire graph, such as load a graph or save a graph and its layout data. The "Edit" menu defines commands that edit objects (vertices and edges) of a graph, such as adding, deleting and setting parameters. The "View" menu defines commands that control the display window. Various options are provided for the user to create customized views. The "Layout" menu defines commands to parameterize layout algorithms. The "Constraint" menu provides interfaces for
the user to set constraint solvers. Users can use this menu to enable/disable layout constraints, solve constraints without invoking layout components, and save/load constraints. The "Constraint" menu also defines commands to open windows for the user to edit constraints.

The left button of the mouse is used to select an object (vertex or edge) in a graph. The "Edit" menu defines command to open windows to edit attributes of the selected objects. The user also can drag the selected vertex on the screen by moving the mouse. The middle button of the mouse is used to move a subgraph on the screen. The right button of the mouse is used to move the entire graph on the screen.

The "Misc" menu defines commands for converting file formats and evaluating layouts and layout algorithms.
Figure 6.14: LYCA’s main window
Chapter 7

Experiments and Examples

This chapter evaluates the layout algorithms introduced in the thesis by experiments and comparison. The first half of the chapter describes two experiments that evaluate the performance of the revised force-directed placement. The second half of the chapter demonstrates the layout capability of the divide-and-conquer approach and the integration of constraints with two complicated examples.

7.1 Comparative Experiments

This section describes the results of two experiments that compare the layouts produced by the original force-directed placement and layouts produced by the revised force-directed placement. The results of the experiments provide an objective evaluation of the performance of the revised algorithm for graphs that contain vertices of different sizes.

7.1.1 Experiment One

The first experiment uses five sample graphs. The sample graphs are selected from the test graph set used in the evaluation experiment in [20]. The widths and heights of vertices in each graph are randomly chosen from the range between 12 to 100. In the revised force-directed placement, the optimal distance between boundaries of vertices is set as 64. In the original force-directed placement, the optimal distance between centers of vertices is set as \(64 + d_{\text{max}}\), where \(d_{\text{max}}\) is the largest width or height of vertices in a graph.

The layout metrics used in the first experiment are:

- The maximal dimension (width or height) of vertices in a graph.
Figure 7.1: Test Graphs
• k. The optimal distance between vertices.

• CPU time (in seconds) for the revised force directed placement to draw a graph on a Sun Sparc-10 workstation.

• The minimal and average distances between boundaries of vertices in a layout.

• The number of edge-crossings.

• The area that the resulting layout covers.

• The minimal, maximal, average edge lengths and the standard deviation of edge lengths.

The layout of each test graph is generated in a single test run. The results of the experiment are given in Table 7.1 through Table 7.5. The actual layouts created by the two algorithms are shown in Figure 7.2 and Figure 7.3.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Revised Algorithm</th>
<th>Original Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max node dimension</td>
<td>86</td>
<td>86</td>
</tr>
<tr>
<td>Optimal distance</td>
<td>64</td>
<td>150</td>
</tr>
<tr>
<td>CPU time</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>No. of crossings</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Area (width x height)</td>
<td>212x199</td>
<td>210x214</td>
</tr>
<tr>
<td>Min distance</td>
<td>39.62</td>
<td>45.4</td>
</tr>
<tr>
<td>Average distance</td>
<td>57.48</td>
<td>81.29</td>
</tr>
<tr>
<td>Min edge length</td>
<td>39.62</td>
<td>45.4</td>
</tr>
<tr>
<td>Max edge length</td>
<td>84.86</td>
<td>117.44</td>
</tr>
<tr>
<td>Average edge length</td>
<td>57.48</td>
<td>81.29</td>
</tr>
<tr>
<td>Standard derivation</td>
<td>19.4</td>
<td>30.9</td>
</tr>
</tbody>
</table>

Table 7.1: Experiment result for G1
Table 7.2: Experiment result for G2

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Revised Algorithm</th>
<th>Original Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max node dimension</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>Optimal distance</td>
<td>64</td>
<td>163</td>
</tr>
<tr>
<td>CPU time</td>
<td>0.47</td>
<td></td>
</tr>
<tr>
<td>No. of crossings</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Area (width x height)</td>
<td>668x502</td>
<td>900x630</td>
</tr>
<tr>
<td>Min distance</td>
<td>12</td>
<td>66.71</td>
</tr>
<tr>
<td>Average distance</td>
<td>219</td>
<td>123.172</td>
</tr>
<tr>
<td>Min edge length</td>
<td>12.0</td>
<td>66.71</td>
</tr>
<tr>
<td>Max edge length</td>
<td>125.71</td>
<td>224.62</td>
</tr>
<tr>
<td>Average edge length</td>
<td>74.04</td>
<td>123.17</td>
</tr>
<tr>
<td>Standard derivation</td>
<td>29.57</td>
<td>43.12</td>
</tr>
</tbody>
</table>

Table 7.3: Experiment result for G3

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Revised Algorithm</th>
<th>Original Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max node dimension</td>
<td>94</td>
<td>94</td>
</tr>
<tr>
<td>Optimal distance</td>
<td>64</td>
<td>158</td>
</tr>
<tr>
<td>CPU time</td>
<td>0.30</td>
<td></td>
</tr>
<tr>
<td>No. of crossings</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Area (width x height)</td>
<td>480x465</td>
<td>686x671</td>
</tr>
<tr>
<td>Min distance</td>
<td>25.32</td>
<td>57.58</td>
</tr>
<tr>
<td>Average distance</td>
<td>196.01</td>
<td>317.27</td>
</tr>
<tr>
<td>Min edge length</td>
<td>25.32</td>
<td>57.58</td>
</tr>
<tr>
<td>Max edge length</td>
<td>120.00</td>
<td>232.96</td>
</tr>
<tr>
<td>Average edge length</td>
<td>73.38</td>
<td>127.98</td>
</tr>
<tr>
<td>Standard derivation</td>
<td>23.94</td>
<td>45.72</td>
</tr>
</tbody>
</table>
Table 7.4: Experiment result for $G_4$

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Revised Algorithm</th>
<th>Original Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max node dimension</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>Optimal distance</td>
<td>64</td>
<td>163</td>
</tr>
<tr>
<td>CPU time</td>
<td>0.35</td>
<td></td>
</tr>
<tr>
<td>No. of crossings</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Area (width x height)</td>
<td>545x514</td>
<td>680x655</td>
</tr>
<tr>
<td>Min distance</td>
<td>43.97</td>
<td>100.5</td>
</tr>
<tr>
<td>Average distance</td>
<td>226.96</td>
<td>351.00</td>
</tr>
<tr>
<td>Min edge length</td>
<td>43.97</td>
<td>100.50</td>
</tr>
<tr>
<td>Max edge length</td>
<td>87.09</td>
<td>180.18</td>
</tr>
<tr>
<td>Average edge length</td>
<td>76.73</td>
<td>134.18</td>
</tr>
<tr>
<td>Standard derivation</td>
<td>9.64</td>
<td>19.29</td>
</tr>
</tbody>
</table>

Table 7.5: Experiment result for $G_5$

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Revised Algorithm</th>
<th>Original Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max node dimension</td>
<td>97</td>
<td>97</td>
</tr>
<tr>
<td>Optimal distance</td>
<td>64</td>
<td>161</td>
</tr>
<tr>
<td>CPU time</td>
<td>0.17</td>
<td></td>
</tr>
<tr>
<td>No. of crossings</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Area (width x height)</td>
<td>443x393</td>
<td>841x833</td>
</tr>
<tr>
<td>Min distance</td>
<td>57.01</td>
<td>146.22</td>
</tr>
<tr>
<td>Average distance</td>
<td>167.32</td>
<td>486.08</td>
</tr>
<tr>
<td>Min edge length</td>
<td>57.01</td>
<td>146.22</td>
</tr>
<tr>
<td>Max edge length</td>
<td>90.74</td>
<td>224.00</td>
</tr>
<tr>
<td>Average edge length</td>
<td>74.53</td>
<td>194.75</td>
</tr>
<tr>
<td>Standard derivation</td>
<td>8.71</td>
<td>22.396</td>
</tr>
</tbody>
</table>
7.1.2 Experiment Two

In the second experiment, 30 runs are executed for the test graph $G5$. In each run of the experiment, the widths and heights of vertices in the graph are randomly chosen from the range between 12 to 100. The optimal distance for the revised algorithm is set as 64. The optimal distance for the original algorithm is set as $64 + d_{\text{max}}$, where $d_{\text{max}}$ is the maximal dimension of vertices in the graph. The layout metrics used in the second experiment are the same as in the first experiment. Table 7.6 presents the experiment result. In the table, the value of each metric is the average value of the metric in the 30 runs.

![Layouts of $G1$ and $G2$](image)
Figure 7.3: Layouts of $G_3$, $G_4$, and $G_5$
Table 7.6: Result of the second experiment

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Revised Algorithm</th>
<th>Original Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max node dimension</td>
<td>97.7</td>
<td>97.7</td>
</tr>
<tr>
<td>Optimal distance</td>
<td>64</td>
<td>97.7 + 64</td>
</tr>
<tr>
<td>No. of crossings</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Area (width x height)</td>
<td>527x525</td>
<td>674x672</td>
</tr>
<tr>
<td>Min distance</td>
<td>60.01</td>
<td>101.88</td>
</tr>
<tr>
<td>Average distance</td>
<td>229.85</td>
<td>358.19</td>
</tr>
<tr>
<td>Average edge length</td>
<td>78.83</td>
<td>138.74</td>
</tr>
<tr>
<td>Standard derivation of edge lengths</td>
<td>6.73</td>
<td>19.51</td>
</tr>
</tbody>
</table>

7.1.3 Analysis

The results of the two experiments show that the revised algorithm can generate more compact layouts than the original algorithm for graphs with vertices of different sizes. In the first experiment, the minimal distance between vertices is 12 in the layout of $G_2$ and is greater than 25 in the layouts of the other four graphs. The average minimal distance between vertices is 60 in the second experiment. This indicates that the revised algorithm also avoids overlaps between vertices to maintain the syntax validation of the resulting layouts. Although the original algorithm can return more compact layouts if a smaller optimal distance is chosen, the algorithm may not guarantee the syntax validation of the resulting layouts. For example, if the user chooses the optimal distance between the centers of vertices as 97 for graph $G_2$, the original algorithm returns a layout with overlapped vertices, as shown in Figure 7.4. Furthermore, reducing the optimal distance between vertices does not improve the distribution of vertices and uniformity of edge lengths, as discussed below.

Comparing with the algorithm in [12], the revised algorithm consistently returns layouts with smaller standard derivations of edge lengths. This indicates that the revised algorithm outperforms the original algorithm in making uniform edge lengths. The edge lengths also reflect the distances between vertices. A small standard derivation of edge lengths thus implies an even distribution of vertices in the plane.

A problem of the revised algorithm is that it may occasionally return layouts in which edges cross with vertices, as shown in Figure 7.2 (b). Force-directed placement
does not consider this issue explicitly. The original algorithm avoids the problem in Figure 7.2 (b) because vertices are positioned sparsely on the plane. In general, the chance that edges cross over vertices will be increased when vertices are compacted into a small area. The problem in Figure 7.2 (b) can be solved by adjusting positions of vertices or bending the edges that cross with vertices manually.

7.2 Examples

This section describes the layout ability of the divide-and-conquer approach and the integration of constraints with two complicated examples. The two example graphs, G4 and G5, are sample graphs for the graph drawing competitions in the 1994 and 1995 international workshops on graph drawing, respectively. They are chosen from graphs used in real application to evaluate the comprehensive layout ability of a layout tool.

The layouts shown in Figure 7.5 and Figure 7.6 are created by Kamada’s spring algorithm [21] using the graph editor GraphEd [20]. As shown in the two figures, the spring algorithm may return poor layouts when drawing complicated graphs.

The layouts created by LYCA are shown in Figure 7.7 and Figure 7.8. The two layouts are created in the following steps:

- Draw the graphs with the Kamada’s spring algorithm [21].
- Identify interesting subgraphs and partition the graph correspondingly. Some edges are removed in this step to simplify the analysis task.
- Define constraints.
• Draw the graphs with LYCA's integrated approach and divide-and-conquer algorithm.

• Minor manual modification to add edges deleted in previous steps and bend long edges.

Obviously the layouts generated by LYCA are easier to understand than the layouts generated by the spring algorithm. In addition, the user can customize the resulting layouts to reflect different visual structures by constraints and partitioning. As discussed in [10], those features are important to generate layouts that will be understood correctly by users.

The two examples show that LYCA's integrated approach and divide-and-conquer algorithm significantly enhance the user's control on automatic layouts and provide powerful layout ability to handle complicated graphs used in real applications.
Figure 7.5: Layout of graph G4, created by spring algorithm
Figure 7.6: Layout of $G_5$, created by spring algorithm
Figure 7.7: Layout of G4, created by LYCA
Figure 7.8: Layout of $G_5$, created by LYCA
Chapter 8
Conclusion

This chapter summarizes the contributions of this thesis and gives suggestions for future work.

8.1 Contribution

The objective of this dissertation is automatically generating customized layouts according to different requirements given by the user and/or applications. To achieve this goal, new layout techniques are introduced and integrated with existing layout techniques. The result of this work is a layout tool called LYCA.

The contribution of this thesis consists of four aspects:

- Improving the layout capability of the force-directed placement and the Sugiyama algorithm.
- Introducing a new divide-and-conquer approach.
- Introducing a new integrated approach.
- Improving the interactive style of layout tools with interface techniques.

8.1.1 Improving Force-directed Placement and Sugiyama Algorithm

Most graphs used in real applications contain vertices of different sizes. Generating compact layouts for those graphs has practical importance. Although the processing power of computers has increased rapidly in recent years, their display power has not kept pace, e.g., compared with old screens used decades ago, the size and resolution
of today's screens are not increased significantly. It is therefore important to generate compact layouts that fit the small screens of today's computers.

The force-directed placement [12] is a successful layout algorithm in producing aesthetically pleasant layouts. The algorithm, however, is designed to draw abstract graphs that contain vertices of same sizes. When drawing graphs with vertices of different sizes, the algorithm may return large layouts or layouts with overlapped vertices.

LYCA overcomes this problem by revising the force formulas in [12] to consider distances between the boundaries of vertices. The revised algorithm adjusts distances between vertices according to the actual shapes and dimensions of the vertices. The experiments described in chapter seven show that the revised algorithm can generate compact layouts for graphs with vertices of different sizes while maintaining the syntax validation of the layouts. The revised algorithm also consistently outperforms the original algorithm in making uniform edge-lengths and distributing vertices evenly.

The Sugiyama algorithm is modified in LYCA in two aspects. The barycentric ordering algorithm used in the second phase of the original algorithm is implemented with layout forces. The fine-tune step is also performed with layout forces. There are two advantages of using forces to implement the Sugiyama algorithm. The first advantage is that the Sugiyama algorithm can be integrated with the constraint solver to support constraints along the X-dimension. This is important for the user to generate customized layouts. Furthermore, the solver and the Sugiyama algorithm can cooperate to resolve competitions between satisfying constraints and improving aesthetics by biasing the forces on vertices. The second advantage of using forces is to straighten long paths in the fine-tune step. LYCA introduces preferential forces in the fine-tune step to push dummy vertices along a long path toward the center of the path. In addition, the user can straighten long paths interactively: LYCA detects portions of long paths that are modified as fairly straight lines by the user and straightens such fairly straight lines. This feature is especially useful when it is difficulty to straighten an entire long path in the resulting layout.
8.1.2 The Divide-and-conquer Approach

A divide-and-conquer algorithm is introduced in this thesis to generate structured layouts. Divide-and-conquer is a powerful way to handle complicated problems. In previous works, divide-and-conquer approaches are mainly used to improve the efficiency of a layout algorithm, e.g., to draw graphs with a large number of vertices [34]. The objective in this work is to enhance the expressive capability of a layout algorithm. The divide-and-conquer algorithm can display proximity, zones, symmetries and shapes of subgraphs. Those features are useful to reflect semantics of graphs and create perceptually organized layouts as described in [26].

The divide-and-conquer algorithm also improves layout quality for complicated graphs. As shown by the examples in chapter seven, the spring algorithms and the force-directed placement may return poor layouts for complicated graphs. With the divide-and-conquer approach, the layout problem is simplified and the layout quality is improved significantly.

A circular dependency problem is inherent in a divide-and-conquer approach: edges between subgraphs depend on subgraph layouts, which in turn depends on edges between subgraphs. If edges between subgraphs are ignored in subgraph layouts, the resulting layouts may have long edges and edge-crossings. Previously proposed divide-and-conquer layout approaches either require manual modifications of the resulting layout or recursive adjustment of subgraph layouts [19, 40]. LYCA overcomes this problem in two ways. It first chooses a good initial configuration as the input to the divide-and-conquer approach using a normal layout algorithm. It then generates subgraphs layouts and the resulting layout simultaneously. During the process, subgraphs are transformed from “white-box” status into “black-box” status gradually. The heuristic effectively reduces edge-crossings and avoids long edges in the resulting layouts. This ensures the layout quality of the divide-and-conquer approach without manually or recursively adjusting subgraph layouts.

8.1.3 Integrated Approach

A trend in graph layout is to integrate different techniques to handle various layout requirements in real applications. LYCA's approach to integration is simple and
efficient. In each layout iteration, LYCA’s solver inputs a layout from the layout component and returns a layout that satisfies constraints. The solver and the layout component also cooperate to detect and remove congestion caused by constraints.

Comparing with the approach of EDGE [42], LYCA’s approach of integration has two advantages. Because LYCA’s solver does not validate a large number of constraints repeatedly, the solver works more efficiently than the solvers of EDGE. In EDGE, the solver breaks algorithm-generated constraints to satisfy user-defined constraints. This may result in layouts which satisfy constraints but violate aesthetic criteria. LYCA’s solver detects barriers caused by constraints and resolves such barriers by the cooperation between the solver and the layout algorithms. This improves the quality of layouts with a reasonable overhead.

Another important integrated approach proposed previously is TreeSnake [30]. TreeSnake uses options provided by a layout algorithm to satisfy constraints. It can return optimal layouts when options are adequate to deal with constraints. Comparing with TreeSnake, LYCA can not guarantee an optimal solution. On the other hand, LYCA’s solver is not restricted by the options of layout algorithms. TreeSnake may not be able to handle a situation in which improving aesthetics and satisfying constraints contradict. In LYCA, the solver attempts to achieve both aesthetics and constraints by the cooperation between the solver and the layout algorithm. If constraints and aesthetics cannot be achieved in the same layout, the solver satisfies constraints by violating aesthetic criteria. Hence LYCA can handle more types of constraints than TreeSnake.

8.1.4 Improving Interactive Styles

One of the objectives of this work is to support the user to control layouts by interacting with the layout algorithm directly. This objective is achieved to a limited extent.

To help the user diagnose layout problems, LYCA visualizes the layout process at two levels of abstract. Animations are mainly used to examine the layout process in a detailed level, while aesthetic measures provide a quick profile of the layout process. LYCA also provides interfaces for the user to stop, rewind, forward, and resume the
layout process. Put together, those features provide the user with a sense of direct manipulation on automatic layouts.

8.1.5 Other Features of LYCA

LYCA also has other interesting features. It provides interfaces for the user to parameterize layout algorithms. The status of constraints are visualized to help the user understand the behavior of the solver. Users also can invoke the constraint solver to preview the effect of a constraint. LYCA runs on SUN Sparc station with X-window environment. There are 53 classes in LYCA's class hierarchy. The entire program contains about 20,000 lines of C++ code. It has been tested in small scale with graphs generated by tools in SERL or graphs used in research papers on graph layout. The layout tool performs well in the experiments. With slight modification, LYCA can be used in actual applications and serve as user interfaces for meraTalk, PROUD, and other tools in SERL.

8.2 Limitations and Future Work

The scope of this work is limited by the following assumptions.

This work does not intend to develop new constraint solving algorithms. Logic programming is an important research field in which many issues remain as challenges. It is out of the scope of this work to develop new algorithms for solving constraints. LYCA only uses existing algorithms to handle constraints in graph layout. It is possible to replace the algorithm used in the current solver with other constraint solving algorithms.

This work does not intend to develop dynamic algorithms or incremental algorithms which construct layouts interactively. Instead, incremental layout in this dissertation means that the layout process is made incremental and reversible. Those features are useful to control automatic layouts. They also improve the interactive style of a layout tool.

This work has several limitations, which also suggest further work.
8.2.1 Partitioning Problems

In LYCA's divide-and-conquer approach, a graph is partitioned into subgraphs manually. This step is often the most difficult and time-consuming step in the entire layout process. The quality of the partition significantly effects the quality of the resulting layout. An important task is thus to automate this task in order to improve the efficiency and quality of the divide-and-conquer approach. The partitioning problems have been investigated in previous researches. An ideal partitioning algorithm should be able to partition a graph into subgraphs according to the semantics of the graph and/or the graph theoretical properties of the graph, e.g., symmetries or connectivity.

8.2.2 Enhancing the Divide-and-conquer Algorithm

Currently the divide-and-conquer algorithm only can handle graphs with one level of division, i.e., graph and subgraphs. It is desired to enhance the algorithm to handle graphs with multiple levels of division, e.g., a large graph may be partitioned into several subgraphs, which in turn may be divided into subgraphs. Because many problems in real applications can be represented as nested structures, such a recursive divide-and-conquer approach will allow the user to model those problems in a more natural style. It also makes the divide-and-conquer approach more expressive.

A possible way to achieve this goal is to divide the layout time into phases according to the number of the levels of division in a graph. The layout algorithm then transforms subgraphs into "black-box" status in different phases, e.g., subgraphs at the lowest level are transformed into "black-boxes" in the first phase and subgraphs at the highest level are transformed into "black-boxes" in the last phase of the layout process.

8.2.3 Space Creation and Compaction

In LYCA, the revised force-directed placement is used to draw graphs with vertices of different sizes. Another potential application of the algorithm is to perform space compaction when a subgraph is collapsed into a single composite vertex or space creation when a composite vertex in a graph is expanded into a subgraph. This
problem has been investigated in [38, 36]. One possible way to do so is to reduce or increase the size of a vertex or subgraph gradually while adjusting the layout with the revised force-directed placement incrementally.

8.2.4 Improving Performance

In LYCA, the solver and layout algorithms cooperate to resolve barriers caused by solving constraints. It was found that the integrated approach of LYCA works reasonably well in two situations: when a small number of constraints are defined or a large number of constraints are defined. In the former case, the layout algorithm builds up the frame of the resulting layout and the solver performs minor adjustment of the layout generated by the layout algorithm to satisfy constraints. In the later case, the solver builds up the frame of the resulting layout and the layout algorithm “beautifies” the layout created by the solver into a nice-looking one.

When neither the solver nor the layout algorithm can dominate the layout process, LYCA may return poor layouts that require manual modification. Because optimizing aesthetics and satisfying constraints simultaneously is in general intractable, there is no trivial solution to this problem. However, it is interesting to investigate how to improve the layout quality using some known techniques, e.g., local temperature [13] or adding vertices in a certain order [55].

In the current implementation, time performance is not the first concern. Although LYCA is efficient enough as a realistic application, the performance of the current implementation of LYCA can be improved further.

8.2.5 Dynamic Interaction and Knowledge-based Layout

The interaction between the user and the layout algorithm is at a primitive level. It was found in the experiments that animation is not as useful as expected. The control interfaces of LYCA’s layout monitor are mainly used as a powerful “undo” mechanism. A possible way to improve LYCA is to use more advanced program visualization techniques. Another interesting and important work for the future is to integrate the user’s manual layout with automatic layout techniques intelligently and dynamically. For example, a layout tool that can learn layout preference from
user's editing, or the user can intervene in the layout process to set options of layout operations or change methods of solving constraints.
Appendix A

LYCA Documentation

A.1 Operational Manual of LYCA

This is a simplified operational manual for LYCA. The manual explains the major commands and windows of LYCA. If a command or a window has been described in the dissertation, it is omitted or explained only briefly in the manual.

A.1.1 Overview and Environment

LYCA is a tool that can generate graph layouts automatically. Directed graphs are drawn as hierarchy graphs. Undirected graphs are drawn in straight-line style. Constraints can be defined to generate customized layouts.

LYCA runs on a SUN Sparc station with color monitor and X-window environment. The current version of LYCA accepts graphs in MERA format, e.g., graphs generated by meraTalk, PROUD or other tools in SERL. The output of LYCA is also in MERA format and can be used as input to meraTalk or other tools in SERL.

A.1.2 Icon Commands

The main window of LYCA is shown in Figure 6.14. At the left-upper corner of the main window are three buttons (see Figure A.1). The three buttons represent layout commands defined in LYCA. The first button “Undirected” is to draw a graph as an undirected graph. The second button “Directed” is to draw a graph as a directed graph. The third button “Fine-tune” is to fine-tune the current layout.

Below the layout buttons is a panel of icons that represent different editing commands. The two commands shown in Figure A.2 rotate a graph in a clockwise direction or a counter-clockwise direction.
The left command in Figure A.3 inserts a breaking point in the middle of an edge so the user can bend the edge by moving the breaking point. The right command removes a breaking point in an edge. The two commands are useful to draw graphs in mixed convention interactively.

The left command in Figure A.4 detects fairly straight long paths in the current layout and straightens such paths. The right command in Figure A.4 moves the current layout to the top-left corner of the drawing window.

The two commands in Figure A.5 add vertices and edges to a graph. To define a new edge, the user first clicks the "Edge" icon, LYCA then prompts the user to select the source and target vertices of the edge.

A pallet of colors is located below the editing buttons (see Figure 6.14). The user can assign colors for constraints using the color pallet.
A.1.3 Menu Commands

There are five menus in LYCA's menu bar, as shown in Figure A.6.

Figure A.6: Menu bar in LYCA's main window

File Menu

The "File" menu in the menu bar defines commands on graph files:

File Menu:
Load
Save
Save as ...
Close
Quit

The "Load" command opens a window for the user to select and load a graph, as shown in Figure A.7. If a graph has a layout file, LYCA automatically loads the corresponding layout file when the graph is loaded. Currently LYCA only accepts graph files in MERA's format. Layout files are in text format and can be edited by the user with any text editor.

The "Save" and "Save as" commands save the current graph into a MERA file. They also save all the layout information defined on the graph in its layout file. The layout file has the same name as the graph file. The suffix of a graph file is "cdf." The suffix of a layout file is "lyca."
The "Close" file closes the current graph file and its layout file. The "Quit" command closes the current file and quits LYCA.

**Edit Menu**

The "Edit" menu defines commands to edit objects of a graph.

*Edit Menu:*
- Undo
- Edit
- Add Vertex
- Add Edge
- Copy
- Cut
To edit an object (a vertex or an edge), the user first double clicks the mouse pointer on the object to select it. A selected object is highlighted in red color. The “Edit” command opens an editing window for each selected object. User can browse or edit attributes of the object. Figure A.8 shows the editing window for a vertex with the name “Up.” The user can set the name, dimensions, weight and temperature of the vertex in the window.

The “Add Vertex” command adds a vertex to the current graph. The “Add Edge” command adds an edge to the current graph. To define an edge, the user first selects a vertex as the source of the edge. When the user chooses the “Add Edge” command, LYCA asks the user to select another vertex as the target of the edge.

The “Copy” command copies the current vertex into LYCA’s editing buffer. The “Paste” command adds the vertex in LYCA’s editing buffer to the current graph. The user needs to rename the pasted vertex to avoid duplicated names. The ”Delete” command deletes the selected object. The “Undo” command cancels the last editing operation on the current graph.
View Menu

The “View” menu contains commands that control the display of LYCA.

View Menu:

- Refresh
- Constraint
- Subgraph
- Color
- Monitor
- Fonts
- Size

The “Refresh” command clears LYCA’s drawing window and redraws the current graph. The “constraint” command sets the option of displaying the status of vertices. Each constrained vertex is in one of the three status: “Reference,” “Satisfied” or “Unsatisfied.” A vertex is in “Reference” status if the solver never changes its position to solve constraints. A vertex is in “Satisfied” status if all constraints defined on it are satisfied in the current layout and therefore the solver is less likely to change its position to solve constraints. A vertex is in “Unsatisfied” status if there is at least one constraint defined on it that is not satisfied and the solver will definitely change the position of the vertex to satisfy the constraint. Status of a vertex is indicated by the border around the icon of the vertex. A vertex in “Reference” status is displayed with a border in blue color. A vertex in “Satisfied” status is displayed with a border in green color. A vertex in “Unsatisfied” status is displayed with a border in red color. The user can change the colors of constraints with the color pallet in the main window. Status of vertices are updated dynamically as the user edits a graph or LYCA draws a graph with animation.

The “subgraph” command in the “View” menu sets the option of displaying zones of subgraphs. A zone of a subgraph is defined as the minimal rectangle that covers the subgraph. If this option is set as “on,” LYCA displays the zone of a subgraph as a border around it. If this option is set as “off,” zones of subgraphs are not displayed. Zones of subgraphs are updated when the user edits a graph or LYCA draws a graph with animation. Figure A.9 gives two displays with different options of displaying subgraph zones.
The "Monitor" command in the "View" menu pops up the monitor window to visualize how aesthetics are improved in the layout process. Several aesthetics can be checked, each one is represented as one plot in the drawing area of the monitor window.

The monitor window also provides interfaces to control the layout process. There are six buttons below the plot area of the monitor window, as shown in Figure A.10. The first two buttons from the left rewind and forward the layout animation, respectively. The third button from the left pauses the layout process for the user to edit the intermediate results. The fourth button from the left clears the plot area. The rightmost button resumes the layout process if it is paused by the user.
Layout Menu

The "Layout" menu provides interfaces to parameterize layout algorithms:

Layout Menu:
Sugiyama Algorithm
Force-directed placement
Fine-tune

Each command opens a window for the user to set parameters of the corresponding layout algorithm.

The window for the force-directed placement algorithm is shown in Figure A.11. On the top row of the window are four small windows, each provides two radio buttons to set a layout option. The animation option tells LYCA whether or not to visualize the layout process as animations. The cluster option controls the cluster constraints. When it is set as "Off," LYCA disables cluster constraints. When this option is set as "On," LYCA uses a divide-and-conquer algorithm to process cluster constraints. The grid option forces each vertex to be placed on a grid point. The cool option allows the user to choose different cooling functions to control the layout process. There are seven sliders below the small windows. The "Max Iteration" slider sets the maxim number of iterations of the layout algorithm. The "Initial temperature" slider sets the initial temperature of the force-directed placement algorithm. This option should be used in combination with the cool option. The "Optimal distance" slider sets the optimal distance between vertices. The "Optimal Meta-distance" sets the optimal distance between subgraphs. The unit for all distances is one pixel on the screen. The "Cool rate" and "Animation interval" sliders set the speeds of cooling and animation, respectively.

The window that sets the parameters of the Sugiyama style algorithm is shown in Figure A.12. The animation and grid options are the same as in the force-directed placement algorithm. The user can change height of layers using the "Layer Height" scale bar. The "Optimal Gap" option sets the optimal gap between the vertices in the same layer, e.g., if the distance between two neighbor vertices is shorter than the optimal gap, the fine-tune algorithm will push the two vertices away from each other using forces. As explained before, LYCA uses preferential forces to push vertices
**Figure A.11:** Window for force-directed placement algorithm
along a long path toward the center of the path. The strength of the preferential forces for straightening long paths can be set with the “Preferential Force” scale bar.

The window that sets the parameters of the finetune algorithm is shown in Figure A.13. Since the fine-tune algorithm is a customized version of the force-directed placement algorithm, all the parameters for the fine-tune algorithm are the same as in the force-directed placement algorithm. The difference is that some options for the force-directed placement algorithm are disabled in the finetune algorithm such that the fine-tune algorithm will not change a layout drastically.
Figure A.13: Window for the finetune algorithm
Constraint Menu

Commands on layout constraints are defined in the “Constraint” menu, including:

Constraint Menu:
- Constraints
- Load constraints
- Save constraints
- Save constraints as
- Solve constraints
- Enable constraints
- Disable constraints

The “Constraints” command opens a window for the user to edit constraints. In the current implementation, LYCA opens two windows to define constraints in the X-Dimension and the Y-Dimension separately. Constraints are displayed as a list of text specifications in the constraint windows, as shown in Figure A.14. To add a constraint, the user types in the constraint in the text field and presses the “Add” button in the constraint window. If a constraint is syntactically correctly defined and does not conflict with existing constraints, it is accepted. Otherwise, LYCA displays a list of constraints which conflict with the newly defined constraints and reports which constraints are removed to resolve the conflict.

To delete a constraint, the user first selects a constraint by clicking the mouse pointer on it. The user then presses the “Delete” button in the constraint window to delete the selected constraint.

The “Save” command in the “Constraint” menu saves layout constraints for the current graph in a text file. The “Load” command loads a set of constraints defined in a text file.

The “Solve constraints” command invokes LYCA’s solver to solve constraints. But the command does not invoke layout algorithms to improve aesthetics, as shown in Figure A.15 and Figure A.16.

The “Enable constraints” and “Disable constraints” are used to enable and disable layout constraints. When constraints are disabled, LYCA’s solve does not solve constraints.
V-Constraints:

Name rel [Name [{-1+} Constant]]

n15 = n17 + 64
n12 = n18
n16 = n15

Figure A.14: Window for constraints

Figure A.15: Before selecting “Solve constraint” command
Figure A.16: After selecting “Solve constraint” command
A.2 Constraint Language

LYCA supports three kinds of constraints: absolute constraints, relative constraints, and cluster constraints. The usage of the constraints has been explained in the chapter on the constraint solver. This section defines the syntax of LYCA’s constraint language.

```
constraint-list : Constraint_Begin constraint* Constraint_End
constraint : absolute | relative | cluster
absolute : name[.dim] FX [: P=priority]
relative : name[.dim] rel name[.dim]
          [adjust] [: P=priority] [: R=method]
cluster : Cluster( name* ) [: SN=name]
rel : = | > | < | CT | NB
adjust : + integer | - integer
dim : x | y
priority : integer
method : D | B
```

A constraint list is a list of constraints. Each constraint can be an absolute constraint, a relative constraint, or a cluster constraint. An absolute constraint fixes a vertex in one or two dimensions. A relative constraint defines a spatial relation between a pair of vertices. A cluster constraint groups several vertices into a subgraph. Constraints can be defined along either the X-Dimension or the Y-Dimension. If the field “dim” is omitted, the constraint applies in both X and Y dimensions. For relative constraints, the user can choose the vertex the position of which should be changed by the LYCA to satisfy the constraint. By default, LYCA always attempts to solve a constraint by changing the position of the first vertex in the constraint. If the user sets the “R” option to “B”, LYCA first tries to solve a constraint by changing the position of the first vertex. If this conflicts with other constraints, LYCA then tries to change the position of the second vertex to solve the constraint. If the “R” option is set to “D”, LYCA solves the constraint by changing the position of the first vertex in the constraint. Priority should be an integer between 1 and 5. The default priority is 3.
Bibliography


