Master Thesis

Software Visualization via Hierarchic Graphs

Alfred Schuhmacher
February 2, 2015

Reviewers: Dr. Martin Nöllenburg
Prof. Dr. Peter Sanders
Advisors: Dr. Martin Nöllenburg
Dr. Ignaz Rutter

Institute of Theoretical Informatics
Faculty of Informatics
Karlsruhe Institute of Technology

Name:	Alfred Schuhmacher
Student ID:	1633952
Pursued Degree:	Master of Science
Date of Submission:	February 2, 2015
Editing Time:	6 Months

Acknowledgements

First, I would like to thank Dr. Martin Nöllenburg and Dr. Ignaz Rutter. They did not only support me competently during this thesis, but also allowed me to work on a topic that I chose for myself and supervised me even though I studied abroad.

Furthermore, I would like to thank the Softwareschneiderei and, in particular, Daniel Lindner for assisting me in my research by providing me with ideas and real-world data for my experiments.

Contact Information

Author:	*University:*
Alfred Schuhmacher	Karlsruhe Institute of Technology
thesis@zacherl.it	Kaiserstraße 12
	76131 Karlsruhe, Germany
	Phone: +49 721 608-0
	Fax: +49 721 608-44290
	Email: info@kit.edu
	http://www.kit.edu/

Statement of Authorship

Abstract

Code, the textual representation of software, can be read and modified by humans and is suitable for mapping the logic of a program in a way processable by computers. However, it lacks scalability since it does not convey a broader picture, it is insufficient for the initial exploration of software as well as the analysis of its overall condition.

We provide a general visualization for software that covers its structure and supports both initial understanding and further maintenance. Our basic concept is derived from the map metaphor: We interpret software artifacts as regions on a plane, emphasize their hierarchy by nesting these regions and picture relationships as roads connecting the regions. Geographic maps are scalable and navigable, often interactive and highly extensible. Hence, a software visualization mimicking their behavior gains similar advantages.

We represent software with hierarchic graphs for their capabilities of reproducing the complex structures in software accurately. We elaborate the requirements of a generic software visualization based on the map metaphor, define a layout problem for software graphs and introduce an algorithm to solve it. We employ force-directed methods to place vertices, exploit the hierarchy of the graphs to bundle edges and use a geometric heuristic to route edges. The bundling and routing of edges enables us to create a visualization depicting both general relationships and fine details.

We provide an effective solution for visualizing software projects that preserves their internal structures, enables its exploration and can be easily extended. We also developed a prototype for basic tooling and integrated it into the IDE Eclipse. Furthermore, our approach is also applicable to hierarchic graphs beyond software.

Zusammenfassung

Quellcode, die textuelle Repräsentation von Software, kann von Menschen gelesen und verändert werden und bildet die Logik von Programmen so ab, dass Computer sie verarbeiten können. Text ist aber nur beschränkt dazu geeignet, ein vereinfachendes Bild zu liefern, und erleichtert weder den Einstieg in Software noch die Analyse ihres Gesamtzustands.

Wir stellen eine allgemeine Visualisierung von Software vor, welche ihre Struktur berücksicht und sowohl den Einstieg als auch die weitere Wartung unterstützt. Unsere Grundidee stützt sich auf Landkarten: Wir interpretieren Softwareartefakte als Regionen auf einer Ebene, betonen ihre Hierarchie, indem wir die Elemente ineinander platzieren, und bilden Beziehungen als Straßen ab, die die verschiedenen Regionen verknüpfen. Geographische Karten sind skalierbar und navigierbar, häufig interaktiv und sie können gut erweitert werden; wir gehen davon aus, dass eine Visualisierung, die solche Karten imitiert, ähnliche Vorzüge aufweist.

Wir repräsentieren Software durch hierarchische Graphen, da sie in der Lage sind, die komplexen Strukturen in Software genau abzubilden. Wir arbeiten die Anforderungen einer allgemeinen Softwarevisualisierung aus, definieren ein Layout-Problem und stellen einen Algorithmus zur Lösung desselben vor. Wir verwenden kräftebasierte Methoden zur Platzierung von Knoten, nutzen die Hierarchie der Graphen aus, um Kanten zu bündeln, und berechnen mithilfe einer geometrischen Heuristik Routen für die Kanten. Letzteres ermöglicht uns, eine Zeichnung zu erstellen, die sowohl allgemeine Beziehungen als auch Details wiedergibt.

Wir bieten eine effektive Lösung zur Visualisierung von Softwareprojekten, die deren interne Strukturen bewahrt, ihre Untersuchung ermöglicht und die leicht erweitert werden kann. Zudem haben wir einen Prototypen für ein entsprechendes Werkzeug entwickelt und in die Entwicklungsumgebung Eclipse integriert. Weiterhin ist unser Ansatz auch auf andere hierarchische Graphen jenseits von Software anwendbar.

Contents

1 Introduction **1**

1.1 Related Work 2

1.2 Outline 6

2 Problem Definition **7**

2.1 Software Structure 7

2.2 Graph Model 8

2.3 Visualization 11

2.4 Summary 18

3 Algorithm **21**

3.1 Concept 21

3.2 Graph Transformation 27

3.3 Initial Layout 30

3.4 Vertex Layout 35

3.5 Edge Layout 43

3.6 Conclusion 56

4 Parameter Studies **59**

4.1 Attraction and Repulsion 59

4.2 Gravity 62

4.3 Number of Iterations and Temperature 62

4.4 Summary 65

5 Evaluation **67**

5.1 Results 67

5.2 Future Work 71

6 Conclusion **77**

Bibliography **79**

List of Figures

1.1 Software visualization based on the map metaphor. 2
1.2 Software visualized as a city [46]. 3
1.3 Various software graph visualization approaches. 4
1.4 Visualizations emphasizing the graph hierarchy. 5

2.1 Examples for relationships in code. 8
2.2 Representative vertex in a complex hierarchy. 10
2.3 Reference graph corresponding to Figure 2.4. 11
2.4 Example code from a small software project. 12
2.5 Simple graph layout. 13
2.6 Example layout for a hierarchic reference graph. 14
2.7 Suboptimal vertex placement. 15
2.8 Example node shapes. 15
2.9 Compaction of vertices. 16
2.10 Bundling edges. 17
2.11 Routing edges. 17
2.12 Manual layout for a simple formula parser. 19
2.13 Manual layout for the refactored formula parser. 20

3.1 A partial layout. 22
3.2 Approximation of the position of external vertices. 24
3.3 Parent layout deteriorating the edge routing. 25
3.4 Graph with nodes and segmented edges. 28
3.5 Segmentation of a single edge. 29
3.6 Segmentation and bundling of two edges. 30
3.7 Edges (red) and their projections onto the hierarchy (blue). 31
3.8 Selecting the Jordan center. 33
3.9 Space estimation for child vertices. 34
3.10 Preferred radius for a parent vertex. 37
3.11 Preferred distance between two vertices. 40
3.12 Vertex distance and course of the force function. 42
3.13 Node positions for root segments. 46
3.14 Edge projection and segment trees. 47
3.15 Positions for non-root nodes. 47
3.16 Transition between single segments and a thicker bundle. 48
3.17 Rectangle for a segment $s = (n_u, n_v)$. 49
3.18 Determining the segment offset. 50

3.19 Sorting nodes radially. 50
3.20 Critical points for a segment. 51
3.21 New collisions after routing a segment. 51
3.22 Overlapping neighboring segments. 51
3.23 Unrelated segments joining segments that will be removed. 53
3.24 Segment tree for an exit node. 53
3.25 Merge node computation. 53
3.26 Inevitable crossings. 55
3.27 Edges within a segment. 55
3.28 Segment transitions. 56
3.29 Crossings between multiple edges. 56
3.30 Edges originating at the same vertex. 57

4.1 Radius parameter configurations. 60
4.2 Distance parameter configurations. 61
4.3 Compaction by gravity. 62
4.4 Gravity parameter configurations. 63
4.5 Number of iterations. 64

5.1 Detailed section of the refactored layout. 67
5.2 Automatic layout of the formula parser. 68
5.3 Project A. 69
5.4 Project B. 70
5.5 Project C. 70
5.6 Tool for exploring a software graph. 71
5.7 Highlighting of a vertex and its corresponding edges. 72
5.8 Integration of the tool in Eclipse. 72
5.9 Vertex with suboptimal use of space. 73
5.10 Edge bundles overlapping while leaving a vertex. 73
5.11 Class-centered layout for a hierarchic reference graph. 74

List of Algorithms

3.1 Radial tree layout 35
3.2 Global layout algorithm 36
3.3 Resizing of vertices 36
3.4 Fruchterman-Reingold 38
3.5 Partial layout algorithm 44
3.6 Segment routing 48
3.7 Finding criticalities 52

3.8 Crossing removal 54
3.9 Segment routing 54
3.10 Edge order 57

1 Introduction

The development of software is a complex process. Even in a small project with less than a hundred classes and managed by one person, the developer can easily forget the intention of a specific block of code or lose track of the relationships between different parts of the software. Furthermore, it is not uncommon that projects consist of thousands of classes and millions of lines of code, and larger software is created rather by teams than by single developers. For such reasons, software engineers first started to think about how to read source code [13, 41] and later to look for guidelines for readable and understandable code [29]. This is even more important because successful software usually is maintained over a long period of time in order to fix present bugs and to implement new features that are requested by users.

Today, the dominant representation of software is textual: Text is flexible and can map arbitrary constructs, computers can easily process it and humans are able to understand and modify it. However, it is difficult for humans to quickly comprehend a large amount of text, especially if it contains complex algorithms, and there are software projects comparable to books with thousands of pages. Then again, the need to quickly understand large amounts of code occurs frequently when new developers join the team of an existing project and projects are taken over by different teams. In such a situation, the disadvantage of text is its level of detail. It has to contain all the information so that the computer is able to translate it into an executable program, but if one merely tries to grasp the big picture, detailed text is not necessary and actually obstructive.

Modern programming languages offer additional means for structuring software; in Java, one can use packages and classes to group functionality. Nevertheless, this structure is not very visible and a large part of the inherent interrelations such as references or method calls stay hidden in the text. On this account, a variety of tools for visualizing software was created to facilitate its analysis – but we found that many of them pursue particular objectives such as the indication of bugs [12, 28] or other software metrics [5] and an overall visualization is missing.

At this point our research starts: We aim for producing a general, two-dimensional visualization of software that is not only easily understandable, but also applicable to more specific use cases. We do not intend to create a tool that replaces all other visualization techniques, though it should allow both the exploration of a software and basic analyses. We think that most everyday tasks can be covered by such a visualization. Our main idea stems from cartography; we want to generate something similar to a geographic map for software. Such maps are widely known and can be used intuitively, and just as one keeps the map of a city in their mind, this picture can ease the imagination of the software. Geographic maps also support arbitrary scaling, that is, they can show a whole country as well as small villages. Furthermore, they are extensible and it is simple to add support for interaction; for instance, we can superimpose additional layers to display extra information and navigate through the code using the map.

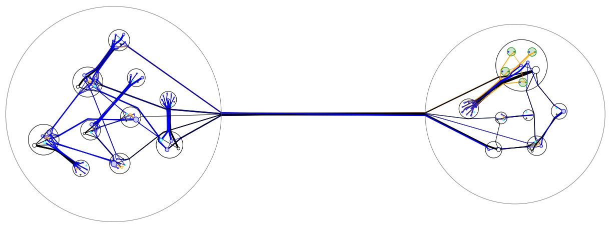


Figure 1.1: Software visualization based on the map metaphor.

We build our visualization on hierarchic graphs for their excellent capabilities of accurately modeling the structure of software. These graphs are able to reproduce the hierarchy occurring in many programming languages; for instance, in Java packages contain other packages and classes, and classes contain methods and possibly nested classes. Moreover, other relationships can be pictured by edges between the vertices representing the software elements. According to the map metaphor, we want to assign a region to each of these vertices and place nested vertices inside of their parents. Then, edges can connect these locations in the form of roads, and just as different routes share roads in reality, we can clean the map up by bundling edges.

An example for such a layout is displayed in figure 1.1. In this work, we focus on the basic visualization and omit the display of subsidiary information. Before giving a short outline of the thesis, we discuss research related to our topic.

1.1 Related Work

Different kinds of software visualization have been extensively studied during the last years; an early example is the approach by Ball and Eick [3] that concentrates on the code and uses simple, but already scalable graphical representations like colored lines or pixels. However, they rather display metrics than the inherent structure of a project.

1.1.1 Software Visualization and the Map Metaphor

A work that already employs the map metaphor is the paper by Young and Munro [48]. They display files and methods of a software project in a virtual, scalable and interactive three-dimensional environment. The software artifacts are pictured by blocks with different colors and dimensions that are chosen according to software metrics like method length. They also include relationships as dependencies by providing a separate visualization.

Panas et al. [39] and Balzer et al. [6] researched similar ideas. The first group uses cities connected by rivers and roads to represent packages and their connections and classes are shown as buildings. They visualize both static and dynamic attributes, for instance, the size of a building indicates the number of lines in a class and method calls result in vehicles moving between different locations. The researchers of the second team restricted themselves to the static structure of software; they picture packages as bubbles and classes as circular disks on which contained methods are placed. Relations are bundled according to the package hierarchy and routed together in three dimensions to reduce visual clutter.

The three-dimensional city metaphor is expanded in a variety of papers leading to results as the one shown in Figure 1.2. Anslow et al. [2] apply the existing description standard X3D to facilitate the creation of suitable tools. Wettel and Lanza [46] focused on improving the metaphor in matters of navigation, interaction and program comprehension. Furthermore, they researched its capabilities regarding program analysis and reverse engineering of software [45]. In contrast, Fittkau et al. [21] and Waller et al. [44] examined its practicability for monitoring the execution of software and provided appropriate tooling. Finally, Balogh and Beszédes [4] even combined the metaphor with Minecraft, a popular computer game, to generate a graphically appealing and easily explorable virtual world.

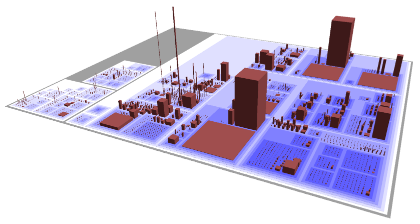


Figure 1.2: Software visualized as a city [46].

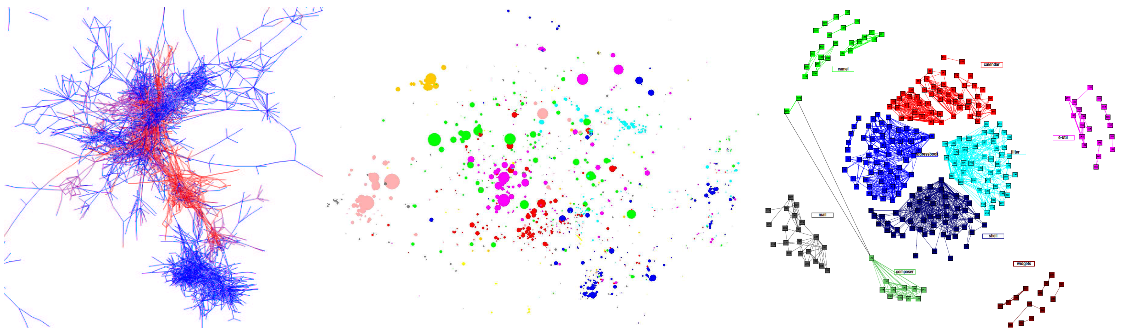
All these approaches offer strong abstraction, a high level of detail and are scalable and extensible. Moreover, they show that the city metaphor is adequate for both generic and specific visualizations. However, even though some of them include the relationships between different software elements, they do not build the visualization around them and in our point of view, the structure induced by these relationships is essential for a general visualization. We also think that a two-dimensional representation is better comprehensible and thus eases the exploration of a software project. An example for such a visualization was given by Kuhn et al. [36]; they mapped software artifacts onto a two-dimensional plane according to their naming. Nevertheless, they did not consider the relations between the artifacts.

1.1.2 Graph-Based Software Visualization

If we forbear from the map metaphor and turn towards graphs, especially in the field of the analysis of software evolution we can find many papers. Typically, vertices of such graphs represent software artifacts like packages and classes and edges model correlations between these elements.

Collberg et al. [11] presented GEVOL, a system that displays relationships as inheritance, method calls and control-flow in form of graphs that change according to modifications in the associated software project. They include detailed information about the general structure of the software, but they only use simple drawings with points and straight lines, and as seen in Figure 1.3a, it can be difficult to spot single components or bigger structures.

In contrast, Beyer [10] introduced a visualization in which vertices are fattened and drawn as spheres in two or three dimensions; an example is given in Figure 1.3b. The vertices are placed near to each other if they are related; in this case, the relationship is determined by how often files are modified together, but it is easy to replace it by, for



(a) Collberg et al. [11] (b) Beyer [10] (c) German [27]

Figure 1.3: Various software graph visualization approaches.

example, method calls. However, he decided not to picture the connections between the different elements. German [27] analyzed the same kind of relationships and, amongst others, provided a visualization illustrated in Figure 1.3c using uniform vertices and straight lines as edges. Like the visualization by Collberg, it tends to become cluttered fast.

Abuthawabeh et al. [1] used a broader input to derive graphs from software and regarded several kinds of relationships including evolutionary coupling. But even though they created interactive visualizations, they focused on facilitating the analysis of specific characteristics instead of giving a general map.

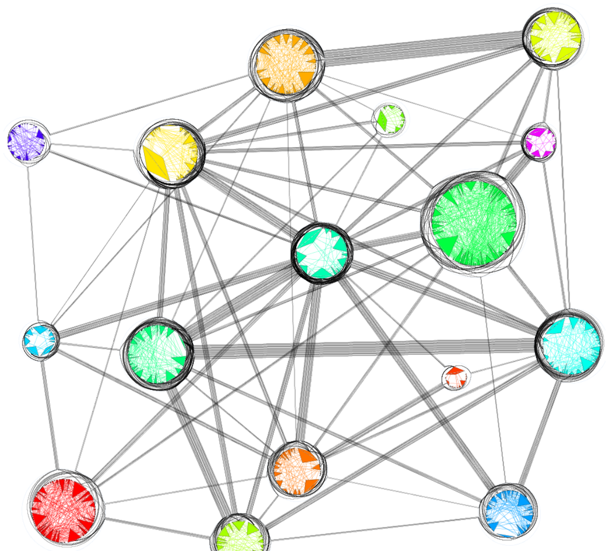
Another interesting approach was taken by Palepu and Jones [38]. They employed the neural network of the brain as a metaphor for visualizing the execution of software. Nevertheless, it is still a rather simple visualization using dots and lines of different thickness to display vertices and edges.

Finally, we want to refer to a work by Reiss and Tarvo [42] that we deem really interesting. They broke the union of single source files by isolating elements like methods or small classes and displaying them in separate so-called code bubbles. These bubbles are linked to each other, for example, if a class contains a method or if a method calls another one. While the visualization is very detailed since it still contains the source code, it improves the presentation of the files in an editor.

Compared to the techniques mentioned above, we follow a more generic approach. For us it is not only important to include all information about structure and relationships in the graph, we also want to display them simultaneously without cluttering the visualization. For this reason, it is necessary to exploit the given structure to organize the drawing. Next, we enumerate some visualization techniques for graphs not related to software development.

1.1.3 Other Graph Visualizations

The map metaphor was also applied to graphs; Xu et al. [47] used it to compute a three-dimensional landscape for multivariate networks. However, they aim to find and highlight clusters of connected vertices that are similar according to additional attributes


(a) Baur and Brandes [8]

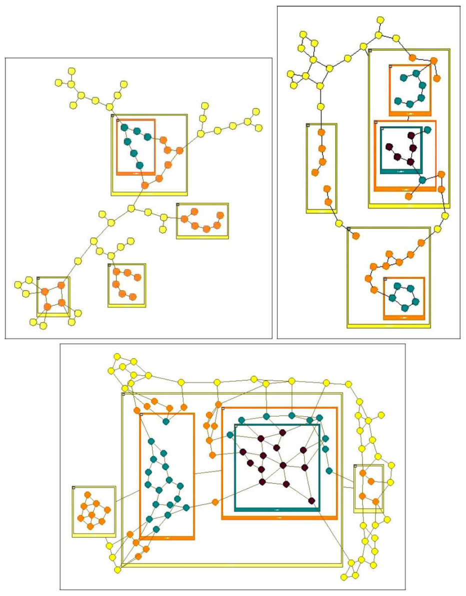

(b) Dogrusoz et al. [15]

Figure 1.4: Visualizations emphasizing the graph hierarchy.

and we intend to find a drawing that nicely reflects the structures given by the relationships of the graph.

Since software is often organized hierarchically, visualizations of clustered graphs are relevant to us. Eades and Feng [19] considered two-dimensional representations as insufficient and developed an algorithm that generates three-dimensional layouts for clustered graphs. They use two-dimensional layouts for the levels in the cluster hierarchy and assemble these layouts in the third dimension. Later, Ho and Hong [31] contributed several linear time algorithms with similar results.

Nevertheless, there are still methods to display these structures in two dimensions. For instance, Baur and Brandes [8] introduced an algorithm for micro/macro layouts. More precisely, for a given graph, the micro graph, and a partitioning, they derive a macro graph by combining the vertices within and the edges between the different partitions. Next, they create a layout for the macro graph with sufficient sizes for vertices and edges so that they can place the micro graphs within the vertices of the macro graph; the edges of the micro graph are routed through the edges of the macro graph. As illustrated in Figure 1.4a, this results in a very tidy layout with nicely bundled edges.

Finally, Dogrusoz et al. [15] proposed a way to visualize clustered graphs with more than two levels in the hierarchy. They use compound vertices and place vertices belonging to a cluster inside of the vertex representing the cluster; an example layout is displayed in Figure 1.4b. However, unlike Baur and Brandes they do not bundle edges between clusters so that these layouts are prone to become cluttered for dense graphs. In this work, we present an approach to visualize software with results resembling a mixture of these two methods.

1.2 Outline

This thesis is structured as follows: In Chapter 2, we start by analyzing the structure of software, elaborate different graph models and evaluate their applicability for representing software. Furthermore, we discuss the requirements for a general visualization of software and give a definition for the layout problem we want to solve.

Chapter 3 deals with how to generate a sufficient layout. First, we examine the advantages and drawbacks of different layout techniques and sketch our algorithm; we also explain what graph transformations are necessary. The algorithm itself is divided into two major steps, the placement of vertices and the bundling and routing of edges. Especially the edge bundling and routing is essential for the eventual clarity of our layout.

In the next part, Chapter 4, we debate the influence of the different parameters and describe the studies we performed in order to find adequate parameter sets for our algorithm.

Afterwards, we present and evaluate our results, indicate points for improvement in our algorithm and provide a perspective to future work in chapter 5, before we finally summarize the thesis and come to a conclusion in Chapter 6.

2 Problem Definition

In this chapter, we accurately describe the problem we intend to solve. First, we shortly deal with the general structure of software, and in the second part we outline different graph models for representing a software project and evaluate their suitability to generate a thorough visualization. Next, we elaborate the requirements of a high-quality general-purpose software visualization using ideas originating from geographic maps. Finally, we summarize our findings and give a brief formal problem definition.

2.1 Software Structure

In this work, we focus on imperative programming languages and use Java as an example. Nevertheless, it should be possible to apply the principles found to most other modern languages. For imperative languages, two major aspects are pictured in source code, namely actions and data; in Java we refer to them as methods and variables.

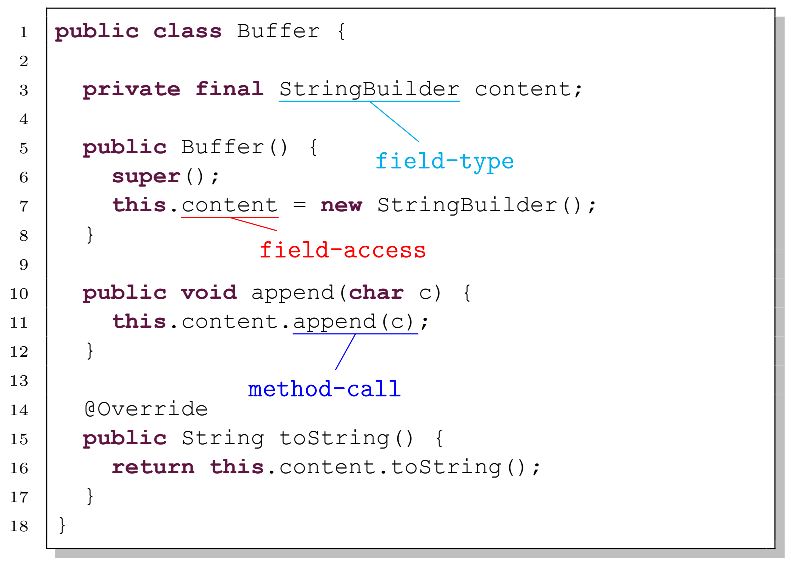
In addition to these base elements, Java and many other programming languages provide some means for structuring software artificially by grouping components like packages, classes and methods. Often, these components already feature some kind of hierarchy as the one for Java depicted below; the element on the left side of the arrow may contain an arbitrary number of elements on the right side. A `type` could be a `class`, an `interface` or an `enum`, and a `procedure` is a placeholder for either a `method`, a `constructor` or an `initializer`.

- `package` → `[package | type]`
- `type` → `[type | method | field]`
- `procedure` → `type`
- `enum` → `enum-constant`
- `[enum-constant` → `class | field | procedure]`

This hierarchy is the part of the structure that is easy to identify in a Java project because it is usually reflected in the folder hierarchy and the indentation of the source code. In contrast, there are other relationships that are less visible since they are only given textually. Below, we list relationships and their involved partners occurring in Java, and an example is found in Figure 2.1. Next, we introduce some graph models that may be used to represent software.

- `method-call`: calling and called `method` or `constructor`

- `method-override`: overriding and overridden `method`
- `constructor-override`: overriding and overridden `constructor`
- `field-access`: accessing `procedure` and accessed `field`
- `field-type`: a `field` and its `type`
- `inheritance`: derived and base `class` or derived and base `interface`
- `implementation`: derived `class` or `enum` and base `interface`
- `reference`: `procedure` and `type`, e. g. for the `type` of a local variable in a `method`



```
public class Buffer {

  private final StringBuilder content;

  public Buffer() {
    super();
    this.content = new StringBuilder();
  }

  public void append(char c) {
    this.content.append(c);
  }

  @Override
  public String toString() {
    return this.content.toString();
  }
}
```

Figure 2.1: Examples for relationships in code.

2.2 Graph Model

The idea to visualize software with the aid of graphs is not new, hence we can build on existing graph models. We focus on three similar models that mainly differ with respect to the relationships between vertices. We start by defining these models, namely the *call graph*, the *reference graph* and the *hierarchic reference graph*, and explain the correlation between software and graph. Furthermore, we give an example for a real mapping from a piece of software to a graph and discuss the pros and cons of the models before choosing one of them.

2.2.1 Call Graph

The call graph of a software is the simplest and probably best-known model that is employed to represent software; it concentrates on procedures or methods and their relationships among themselves. We can construct the call graph $G = (V, E)$ by creating a vertex for each procedure in a project and inserting an edge between two vertices each time a procedure calls another. The graph is not necessarily connected and since one procedure may call another several times, it is a multigraph. It is possible to replace the methods by – for instance in Java – classes or packages, the layer of abstraction can be changed freely. Of course it is necessary to adjust the edges in this case; one could aggregate method calls or use another kind of relationship as class inheritance.

2.2.2 Reference Graph

As stated previously, a software project does not only consist of procedures and there are other kinds of relationships than method calls. Instead, there is usually a multitude of elements working together, which leads us to an extended call graph, the reference graph. Regarding Java, vertices of such a graph do not only represent methods, but also elements like classes, packages and interfaces, and they may be connected by different kinds of edges. Formally, we create a vertex for each software artifact, insert an edge for the respective relations and define two mappings vt and et indicating vertex and edge types to obtain the reference graph $G = (V, E, vt, et)$. Apart from the types, the reference graph does not differ from the call graph; it is also a directed, possibly disconnected multigraph.

$$\begin{aligned} vt &: V \to \{\texttt{package, class, interface, method,...}\} \\ et &: E \to \{\texttt{method-call, field-access, inheritance,...}\} \end{aligned}$$

2.2.3 Hierarchic Reference Graph

While the reference graph already contains all structural information of the software including its hierarchy, it does not distinguish between the hierarchy and other relations. Nevertheless, this hierarchy can be exploited to derive a clustering for the graph, which could help to visualize the graph depending on the method used. We construct the hierarchic reference graph $G = (V, E, vt, et, r, H)$ similar to the simple reference graph. The only difference is that all edges denoting affiliation are added to the hierarchy H instead of the edge set E. An example for such an edge would be a package directly containing a class; we already mentioned the corresponding relations for Java. H is a tree rooted in $r \in V$ that connects all vertices in V, which makes the hierarchic reference graph a cluster graph. It inherits most characteristics of the simple reference graph, that is, it is a directed, not necessarily connected multigraph.

This definition is sufficient to reproduce the relationships in software including the almost arbitrary nesting of types that may happen in programming languages like Java.

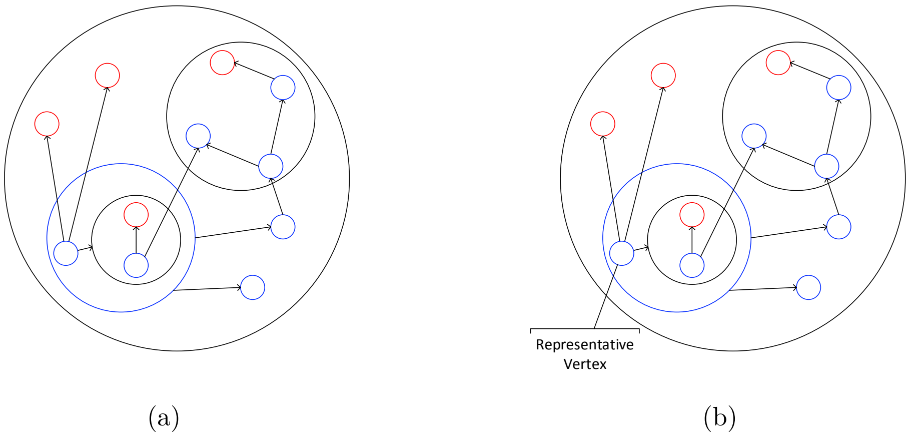


Figure 2.2: Representative vertex in a complex hierarchy.

However, as shown in Figure 2.2a, it is sometimes difficult to visualize edges between child vertices and their ancestors. In this example layout all vertices are drawn as disks and nested vertices are drawn within their parents.

To prevent this, we introduce a representative vertex for each vertex with children. As illustrated in Figure 2.2b, such a vertex is a child of the vertex it represents. It acts as a placeholder for its parent, that is, edges originally leading to the parent instead connect to the representative vertex. It would be possible to differentiate between internal and external edges so that only internal edges are redirected to the placeholder, but this could lead to confusion, so we include all edges. As a result, edges only occur between leaf vertices. Any hierarchic reference graph not fulfilling the representative vertex condition can easily be transformed into a graph respecting the condition.

2.2.4 Code-Graph-Mapping

Before choosing a graph model we want to hint at how to derive a graph from source code by giving an example. We wrote a simple formula parser in Java to determine the requirements for a suitable software visualization. A part of this project consisting of two classes and an interface is displayed in Figure 2.4. The associated reference graph is displayed in Figure 2.3: Types are represented as black vertices, the package is colored dark-green, and methods and fields are painted in blue and red, respectively.

In addition to the extended vertex set, different kinds of relationships are highlighted by changing the appearance of the edges. While field accesses are shown by red arrows, black arrows indicate ownership – for instance methods belonging to classes – and dashed arrows mark interface implementations and overridden methods.

As stated before, the hierarchic and the simple reference graph vary only with regard to their logic representation, so we do not need to adjust the graph in Figure 2.3. However, a standard call graph would consist only of the four blue vertices and the two blue arrows – the remaining information is neglected.

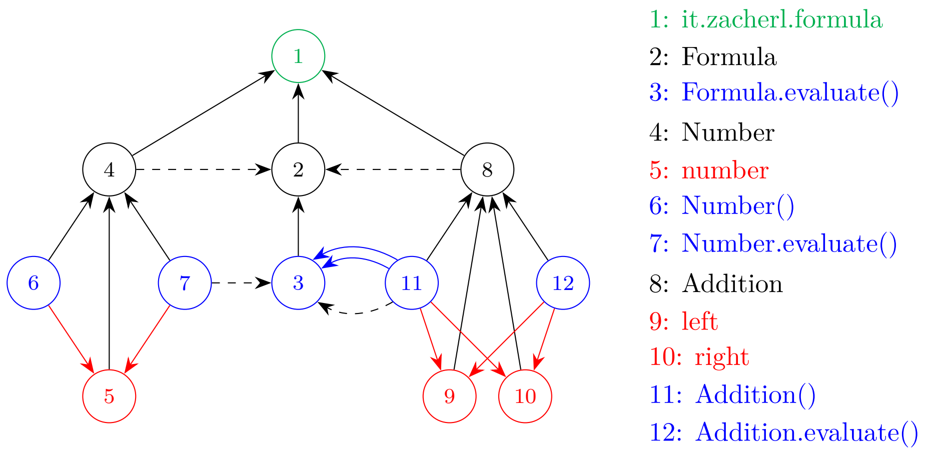


Figure 2.3: Reference graph corresponding to Figure 2.4.

2.2.5 Evaluation

The three graph models differ mainly in two points, namely information density and structure. The call graph focuses on the relationships of the basic elements of a software. Hence, it may deliver the most accurate visualization, but as we could see in the example, it only pictures a part of the relationships and components of a software, so that it is not sufficient as a basis for a general software visualization.

In contrast, the reference graph provides a comprehensive representation of software because the number of different elements and relations is not restricted. However, this model lacks structure: If all relationships are equal, one has to introduce another kind of structure or otherwise the quality of the visualization can decrease drastically for larger graphs. For this reason, we choose the hierarchic reference graph, which emphasizes the artificial hierarchy imposed by software developers.

So far, we defined our graphs close to Java by using real types for vertices and edges, but it is still far from complying with the rules that have to be obeyed. In a valid Java program it is, for instance, impossible to create a method directly within a package, it always has to be within a class, and a method call must originate from another method. However, for us it is only important that we are able to depict a program using a graph – we do not care if it is possible to create a graph representing an invalid piece of software. Furthermore, we do not adjust the layout depending on vertex or edge types, so we can neglect them. We only need to distinguish between hierarchic and non-hierarchic relationships. Hence, from now on we use $G = (V, E, r, H)$ to refer to the hierarchic reference graph.

2.3 Visualization

After deciding which graph model to employ, we can eventually elaborate the requirements for a high-quality software visualization. We first refer to the map metaphor,

```java
package it.zacherl.formula;

public class Number implements Formula {

  private final int number;

  public Number(int number) {
    super();
    this.number = number;
  }

  @Override
  public int evaluate() {
    return this.number;
  }
}
```

```java
package it.zacherl.formula;

public interface Formula {
  int evaluate();
}
```

```java
package it.zacherl.formula;

public class Addition implements Formula {

  private final Formula left;
  private final Formula right;

  public Addition(Formula left, Formula right) {
    super();
    this.left = left;
    this.right = right;
  }

  @Override
  public int evaluate() {
    int left = this.left.evaluate();
    int right = this.right.evaluate();
    return left + right;
  }
}
```

Figure 2.4: Example code from a small software project.

which we deem especially useful for creating well understandable graphical representations of software. In the next step we construct a first layout, point out its deficiencies and elucidate how to improve it. Finally, we show a practical example based on a real project, which we have manually designed according to the requirements given before.

2.3.1 Map Metaphor

We already mentioned the map metaphor in the introduction. Maps have been used by humans for a long time, they are understood immediately by most people and they can be easily applied to many subjects. If we relate to maps, we mean two-dimensional drawings of geographic places such as maps of cities or countries. Usually, they contain locations connected with each other by roads and they are partitioned into colored regions like forests, villages, fields and lakes.

Another point concerns the usage of maps. Sometimes it is helpful to change the scale of a map if one wants to see more or less details. For traditional maps drawn on paper this was achieved by providing different maps, but nowadays it is possible to freely interact with computer maps by zooming in and out or moving the currently displayed section.

Furthermore, maps are often modified in order to suit them to other purposes and they may be more or less abstract; an example is the display of the population of countries by distorting the regions accordingly [26]. Since we are looking for a visualization of software that can be comprehended quickly, it seems promising to build on such a widespread technique.

2.3.2 Basic Layout

Due to the size of most software projects it is hard to visualize them manually. Therefore we have written a parser for mathematical formulas, a small Java program consisting only of two packages and a few classes and unit tests, which we transformed into a graph.

First, we simply tried to draw disks for each vertex, placed them somewhere and connected them with straight lines as displayed in Figure 2.5. Such a visualization reflects the relationships between two elements most accurately since one ignores any existing structure that could separate them. However, it is easy to imagine that such a drawing becomes cluttered for denser graphs with more vertices. It would be possible to indicate characteristics like class affiliation using intelligent coloring, but the capabilities of such a technique are limited because components in the graph can be torn apart. For example, if two methods of the same class are related neither directly nor indirectly, it is likely that they are placed at a great distance.

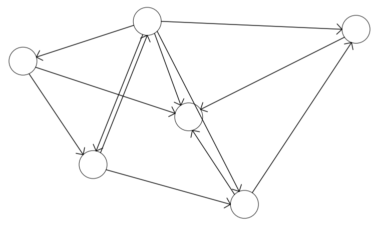


Figure 2.5: Simple graph layout consisting of circular vertices and straight arrows.

This probably occurs in case the natural clustering induced by the method's relationships does not conform to the artificial class structure. On the other hand, the disruption of a project can be seen as a clue about its state. It would be desirable for a neatly programmed software project to feature a nice graph shape. Nevertheless, a tool that creates a general-purpose map for software does not meet its requirements if the generated map is confusing, especially since such a tool might be used as starting point when legacy software projects, which may be of poor quality, are analyzed. In contrast, a good base map can easily be enriched with information such as the inner cohesion of elements by overlaying different views, a technique that is already common for geographic maps.

Next, we applied the map metaphor by regarding the vertices as locations that may contain other locations. By exploiting the hierarchy we constructed a clustering and drew a layout as shown in Figure 2.6. The hierarchy of the software project is emphasized by placing child vertices – for example the methods of a class – inside their parents; hence it serves as a natural zoning. Prior condition is a two-dimensional, non-overlapping and non-touching visualization of the vertices, but this also favors the map metaphor since the regions representing the vertices can be interpreted as states, cities or even parts of cities that are connected by roads. Due to nested regions, such a map inherently supports arbitrary scalability; it is suitable for interactive exploration and features like zooming can be implemented without effort.

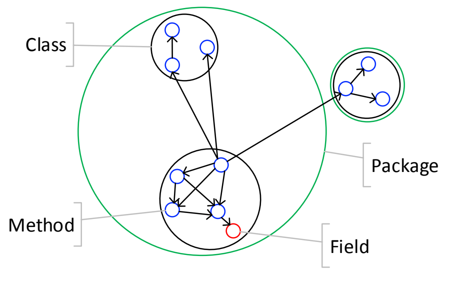


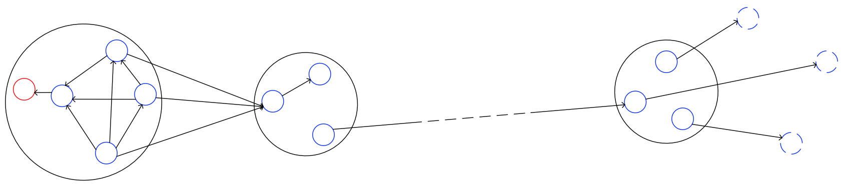
Figure 2.6: Example layout for a hierarchic reference graph.

Nonetheless, there are some drawbacks; the visualization concentrates on an artificial clustering, meaning that vertices can no longer be placed in discretionary fashion and allowing relationships to be pictured suboptimally. In Figure 2.7a, the hierarchy forces one of the method vertices in the middle class to be positioned far away from its only neighbor, even though it is not connected to other vertices in its class. Another example is illustrated in Figure 2.7b, here the strong size difference of vertices leads to a drawing where edges between child vertices are stretched. Furthermore, it is still difficult to track relationships in general as the edges may form a nontransparent entanglement in more complex graphs.

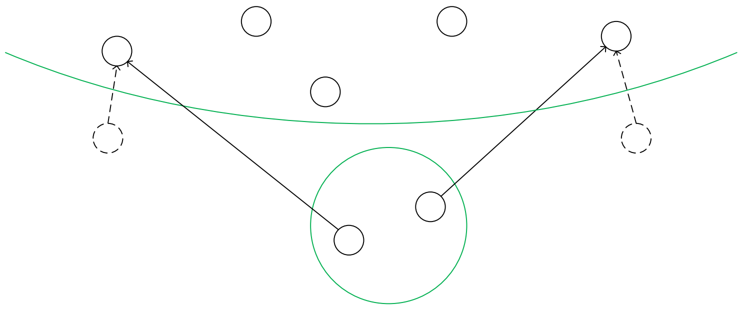
In the next few sections, we address these issues as well as other details and show possibilities for improving the visualization. However, we focus on enhancing the edge layout by bundling and routing edges and retain the nested vertex placement. In our opinion, the advantages of an actively displayed hierarchy prevail compared to its weaknesses regarding the edge layout.

2.3.3 Vertex Visualization

As stated above, vertices must be drawn as two-dimensional objects so that children can be positioned within their parents; Figure 2.8 shows a few examples of vertex shapes. We choose disks since they are simple, space efficient, rotation-invariant geometric elements for which it is easy to detect collisions. Arbitrary polygons or even more complex shapes



(a)



(b)

Figure 2.7: Vertex placement suboptimal regarding the relationships of the vertices.

could lead to interesting visualizations, but also increase the difficulties of designing suitable algorithms.

It is feasible to include specific information such as labels – usually, software elements like classes and methods are named – or manipulate the vertex size according to software metrics. However, even though labels might contain some information about the structure of software [36], we inferred that including text is impeding the construction of the main layout because it consumes too much space. In addition, the map metaphor provides a solution; the labeling of maps is a well-researched problem in cartography that can be applied to the labeling of a graph layout [43]. If an interactive map is provided, it is not only unnecessary, but also confusing to display all labels at all times.

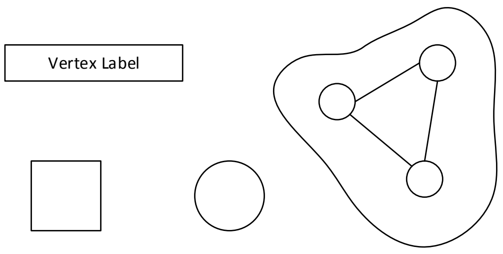


Figure 2.8: Example node shapes.

Likewise, we do not consider additional characteristics such as software metrics, as our goal is to create a general-purpose map rather than a specific visualization. Moreover, the software map can still be altered later on; a good example is the already mentioned distortion of geographical maps known from cartography.

2.3.4 Edge Length and Thickness

In software projects, not all references are of the same importance. For instance, there are basic classes, like `String` in Java, which appear in a huge number of classes, but are not essential for the project itself. Hence, it could be beneficial to thicken and shorten meaningful edges. However, since we use a multigraph, strongly related elements are connected by several edges, and as described in the next section, we can combine these edges to thicker links. Besides, the most limiting factor for the edge length is probably the hierarchy, so it is arguable whether the length of edges can be significantly adjusted.

2.3.5 Crossing Reduction and Compaction

A serious problem in the visualization of graphs derived from software is the sheer number of edges, which can easily ruin the clarity of any layout, and it is especially easy to lose track of edges at crossings with other edges. However, software graphs are not necessarily planar, so in most cases it is not possible to draw them without crossings. Of course, the amount of crossings should be reduced as much as possible, but we touch on this topic only briefly because edge bundling techniques seem to be more promising.

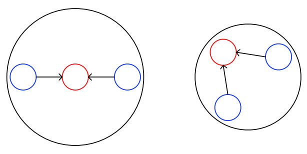


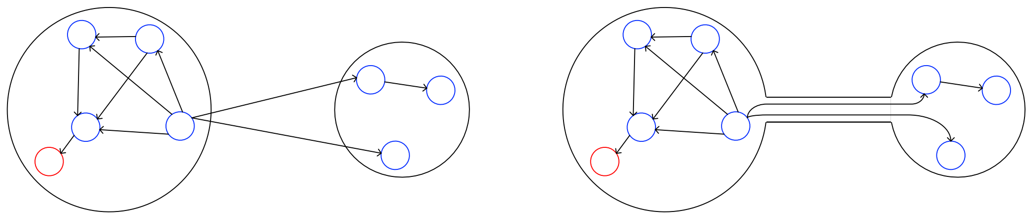
Figure 2.9: Compaction of vertices.

Another difficulty is the space efficient visualization of deep hierarchies. Supposing that nested vertices consume a lot of space, lower levels become hardly visible. For this reason, it is desirable to compact nested vertices as shown in Figure 2.9. In particular, linear structures such as chains of vertices, which consume a lot of space, can be strongly condensed.

2.3.6 Edge Bundling and Routing

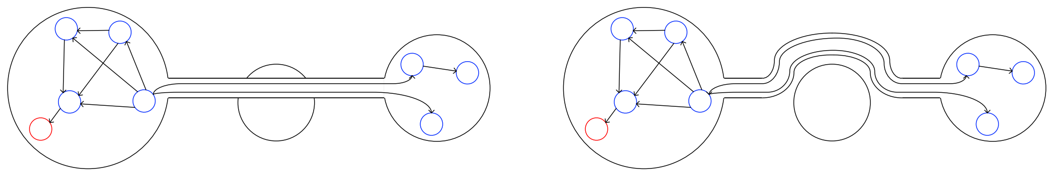
It is also possible to enhance the visualization of edges by bundling them appropriately so that they follow common routes instead and cannot spread over the whole drawing. Again, this supports the map metaphor: First, we only described the aggregation of vertices to cities or regions; at this point edges form wider streets or highways that may converge with other edges or branch out. Furthermore, this simplifies the exploration of major relationships between packages or classes as well as fine connections between methods or fields. In this work, we focus on exploiting the hierarchy to derive a bundling as seen in Figure 2.10 by combining edges that share an end vertex. Additionally, it might be useful to combine even unrelated edges that are located near to each other.

So far, we have only required that vertices are not overlapping with other vertices while ignoring edges colliding with vertices, but if edges are bundled into wider paths,



(a) Drawing edges individually. (b) Grouping edges with similar targets.

Figure 2.10: Bundling edges.



(a) Edge bundle colliding with a vertex. (b) Routing around the obstructing vertex.

Figure 2.11: Routing edges.

it becomes increasingly important to prevent these collisions as well since vertices could be hidden behind edges. Figure 2.11a illustrates a partially concealed vertex; 2.11b shows a possible edge routing avoiding this collision. Precondition for this solution is that there is enough space between vertices for feeding edges through them and, ideally, these spaces grow according to the vertex sizes to scale with the layout size.

The bundling and routing presented in this section should make it easy to recognize relationships between different vertices at several zoom levels. Nevertheless, it is easy to lose track of single edges, so an interactive tool for exploring the map should provide features like highlighting edges and edge bundles.

2.3.7 Example Project

Based on the needs described previously we manually created a layout for our sample project, the formula parser; the result is displayed in Figure 2.12. Even though the layout is clearly improvable, not only relationships between different packages or classes, but also connections within classes and different types of vertices and edges can be recognized easily. Moreover, we even succeeded in optimizing the software itself: For example, when we analyzed the layout, we recognized duplicated code within the `formula`-package on the right side. After rewriting some parts of the code, we were able to create the layout in Figure 2.13, where the package is less cluttered.

2.4 Summary

The hierarchic reference graph is suitable for representing the complex structures occurring in software projects. In the previous sections we informally described the requirements for a visualization of high quality. To conclude, we want to state the problem that is to be solved. Our input is a hierarchic reference graph $G = (V, E, r, H)$. As defined before, such a graph is a possible disconnected multigraph with vertices V and edges E, the hierarchy H is a tree rooted in $r \in V$ that covers all vertices and edges start and end only at leaf vertices. The last condition is very restrictive, but it is easy to transform an arbitrary hierarchic multigraph by introducing representative vertices as described in a previous part of this chapter. In this thesis, we construct a layout for this graph featuring the qualities listed below.

- **Vertex shapes:** Vertices are represented by disks.
- **Edge shapes:** Edges are represented by a series of lines or curves.
- **Hierarchy:** Parent vertices enclose child vertices.
- **Collision-free layout:** No vertex v overlaps with or touches another vertex or an edge that is not incident to v.
- **Edge bundling:** Edges with a similar origin or target should be routed together.
- **Vertex placement:** While related vertices should be placed near to each other, unrelated vertices should be pulled apart. The distance between vertices should scale with the vertex size, that is, larger vertices should be placed further away than smaller ones.
- **Edge routing:** Edges should not deviate strongly from their ideal route and unnecessary crossings should be avoided.

The first four requirements are special in that they are either fulfilled or not, they cannot be optimized. In our layout, they are regarded as hard constraints, that is, we want to achieve them at any cost. However, it is easy to imagine a drawing with these properties. In contrast, the last three requirements are rather vague and it is more difficult to evaluate them; we try to optimize these attributes. In the next chapter, we design an algorithm that generates a layout according to these constraints.

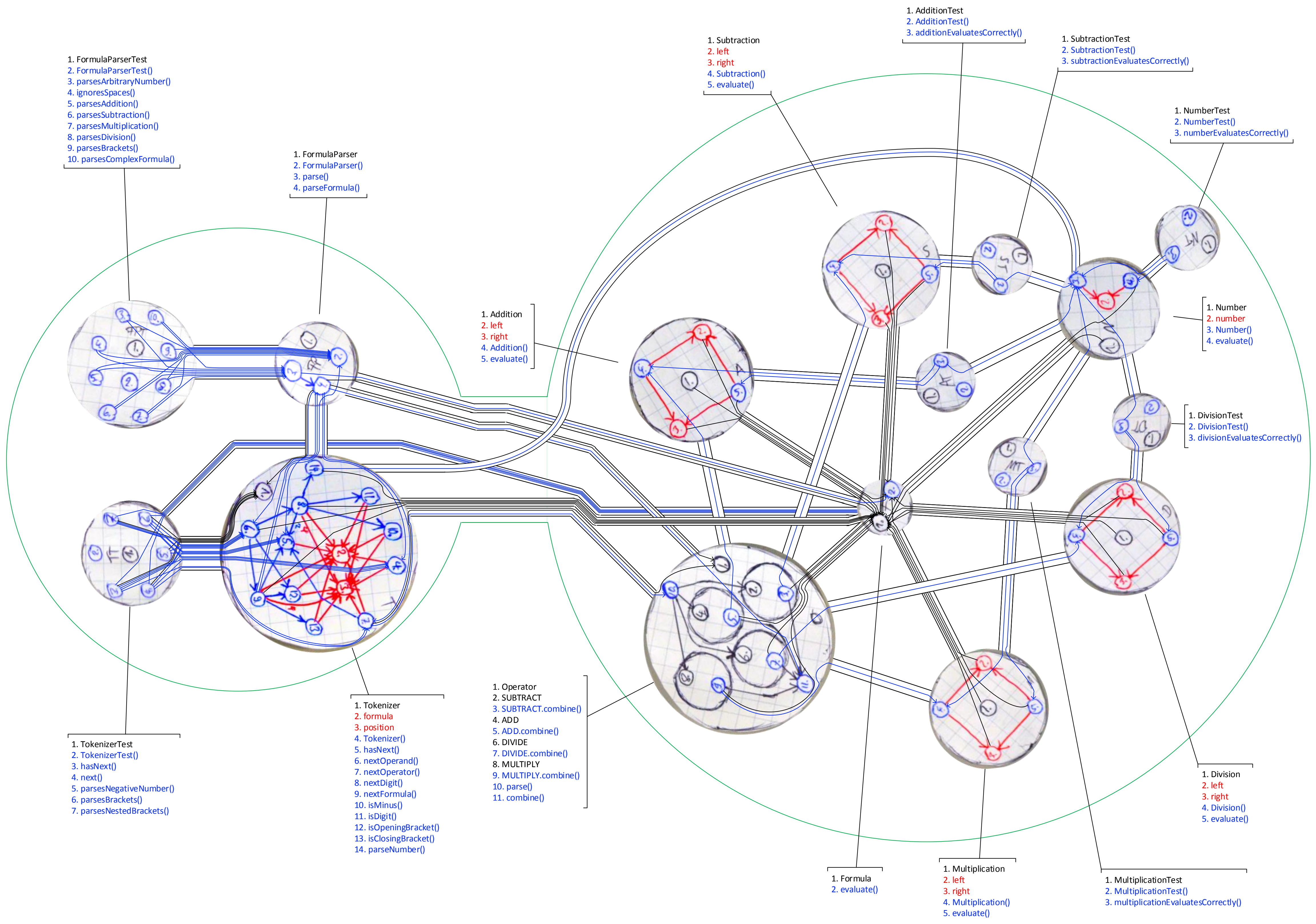


Figure 2.12: Manual layout for a simple formula parser.

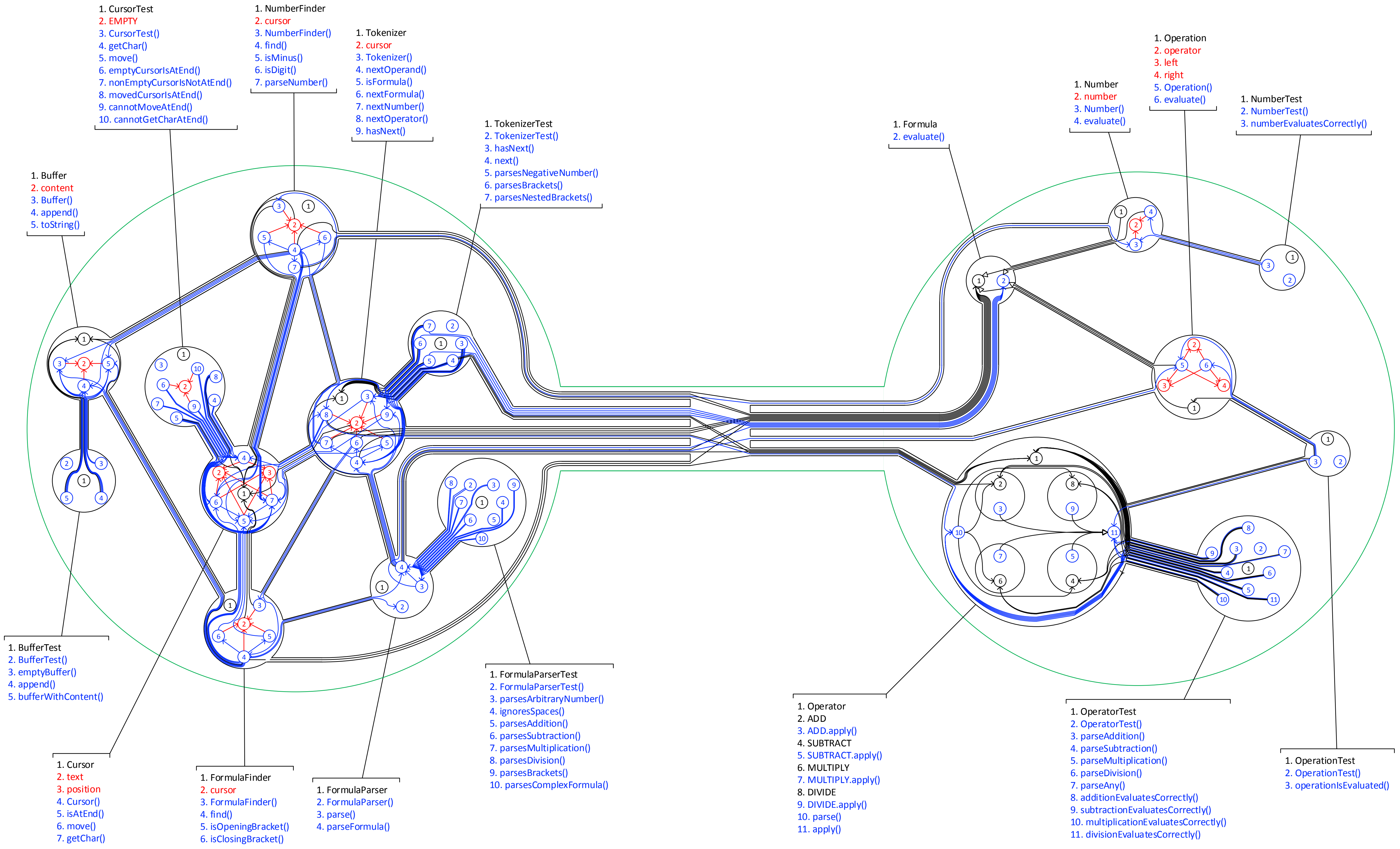


Figure 2.13: Manual layout for the refactored formula parser.

3 Algorithm

In this chapter we design an algorithm that computes a complete layout for an hierarchic reference graph as defined in the previous chapter. First, we introduce our approach, then we describe the steps leading to the final layout. We end by giving a short summary.

3.1 Concept

Graph visualization is a well-researched topic and there already exist many more or less specialized layout methods. In our algorithm we apply force-directed techniques that have been proven suitable for arbitrary graphs. They not only provide layouts of high quality, but are also very flexible; it is easy to adjust them to different requirements [34]. We briefly describe the basic idea behind these algorithms and evaluate a few more sophisticated approaches. Finally, we sketch our algorithm and define terms used in the further work.

3.1.1 Force-Directed Algorithms

As denoted by the name, in force-directed algorithms forces are applied to vertices, usually repulsive forces between all vertex pairs and attractive forces between adjacent vertices. The layout is created by iteratively computing the forces and moving the vertices so that these forces are minimized. Thus, one can achieve a layout in which connected vertices are placed in each other's neighborhood, while unrelated vertices are pulled apart [34]. These techniques are also able to deal with unconnected graphs, for example by using a gravity force pulling all vertices to the center. However, the plain algorithms have a quadratic running time and already for graphs with more than a few hundred vertices the resulting layout may be a local minimum of low quality; many extensions were therefore researched to improve scalability and quality. For instance, simulated annealing is used to reduce the forces over time: At first, strong movements are allowed to avoid local minima, and in the end, the forces tend to zero and the layout stabilizes. Another factor that strongly influences the convergence quality of these algorithms is the initial layout.

3.1.2 Global Methods

Simple force-directed algorithms like the algorithm of Fruchterman and Reingold [23] operate globally, that is, they include all vertices in each iteration. Having said that, our small sample project already contained over one hundred vertices; in real-world projects, tens of thousands are common. As mentioned earlier, there exist more advanced algorithms such as GRIP [24] that can deal with these graphs. Nevertheless, an even more serious problem is the definition of forces that support the hierarchic layout. If all vertices are adjusted at the same time, then both parents and children are moved together. Since children have to be embedded inside their parents, the parents' size depends on the children and their positions, hence the disks representing the parents must be computed dynamically. But due to the simultaneous movement, it becomes increasingly difficult to ensure that not only do children not leave their parent vertex in an arbitrarily deep hierarchy, but also that parent vertices do not overlap. Furthermore, disconnected graphs pose the challenge of placing nonconnected vertices inside a parent. One could solve this issue, for example, by introducing multiple dynamically positioned centers of gravity. However, this would add to the high overall complexity. For these reasons, we concentrate on the local layouts presented in the following sections.

3.1.3 Local Methods

In contrast to global methods, we can derive an intuitive solution for our layout problem by using local methods. In primitive force-directed algorithms repulsive forces are computed between all vertex pairs, which leads to a quadratic running time. A simple optimization is to compute these forces only locally, that is, only for vertex pairs placed near to each other. The respective pairs can be found by partitioning the layout, for example with quadtrees [7].

In our case, it is not possible to apply such a method directly. The main problem of the indicated global approach is to fulfill the constraints given by the hierarchy – and these constraints have to be handled locally as well as globally. Nevertheless, the hierarchy is the characteristic we exploit to create an easy algorithm for our layout task. The most important requirement is that vertex pairs that are not in a parent-child-relationship may not overlap; as already mentioned, parents have to enclose their child vertices. This requirement also implies that the areas covered by the *partial layouts* given by the children of such a vertex pair are independent. With partial layout we refer to the layout given by the children of a specific vertex and the edges between them. As shown in Figure 3.1, the parent vertex, children of the child vertices and other vertices are excluded; the partial layout is a subset of a single layer of the hierarchy. Accordingly, the subgraph covered by the partial layout is denoted by *partial graph.*

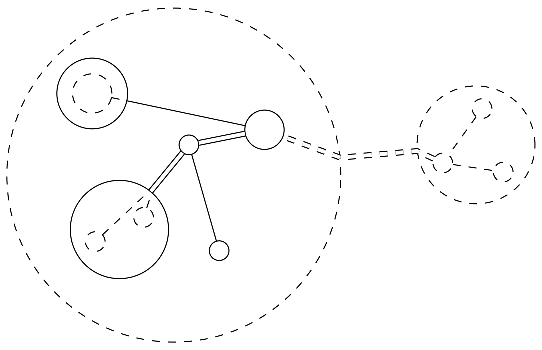


Figure 3.1: A partial layout.

Just as quad trees are employed to partition the layout of a graph, we employ the partial graphs induced by the hierarchy to divide our problem. For each vertex, we individually compute the partial layout, and combine the layouts into one global layout. In these subproblems, we do not need to consider the hierarchy, so that we are able to adjust already existing force-directed algorithms. By regarding only a fraction of the complete graph in each step, we accept that the final layout may be of lower quality; instead, we gain a much simpler model that is still sufficient for producing high-quality layouts. Furthermore, our decision is supported by the software graphs we analyzed: While the overall amount of vertices and edges may be arbitrarily high, the size of the partial layouts seems to be bounded. We examined five real-world projects with vertex numbers between a few thousand and tens of thousands: Even though some partial layouts contained more than a hundred vertices, the average number of children was below ten. Therefore, it is not necessary to use a sophisticated, highly scalable algorithm to create the complete layout.

Now, we start by outlining the algorithm used for computing the partial layout; we explain it in detail later in this chapter. Moreover, we discuss three approaches – bottom-up, top-down and mixed – to compose a complete layout out of the given partial layouts.

Partial Layout Algorithm

As mentioned previously, a partial layout is computed over a non-hierarchic subgraph of the whole software graph. Primitive force-directed algorithms typically use point-shaped vertices connected via direct lines and neglect multiple edges. In our case vertices may contain other vertices, meaning that we have to deal with two-dimensional vertices of different sizes; we assume that the sizes of the vertices in the partial layout are given. In addition, we have to consider that vertices may be connected by more than one edge. However, we show that our directed multigraph can be transformed into an ordinary undirected graph by bundling edges in advance. Still, each edge should be drawn individually, but it is sufficient to regard edge bundles as thick edges and handle the drawing of single edges separately. Along with the requirement to prevent vertices from overlapping or touching, two-dimensional vertices and thick edges pose another challenge: Edges drawn over vertices may conceal vertices or their children, so that it becomes extremely important to route edges. It is also necessary to route edges connecting *internal* vertices with *external* ones, that is, vertices inside and outside the subgraph. These external edges may as well be considered in placing the vertices; it is preferable if vertices with many external edges are placed near the border of their parent vertex.

Bottom-Up Computation

The vertex hierarchy not only partitions the layout of the graph, it can also be used to deduce a natural order in which the partial layouts can be computed, with the most striking orders likely being top-down and bottom-up. Of these, the latter is especially suitable since the partial layout for leaf vertices is empty; it is only necessary to assign a size to them. This size can easily be chosen as a fixed value or according to the degree of the specific vertex. In the next step, it is possible to assign a size to the parent vertex of the leaves by computing their partial layout. Hence, by processing all vertices bottom-up, starting from the leaves and placing vertices relative to their parent vertices, a global

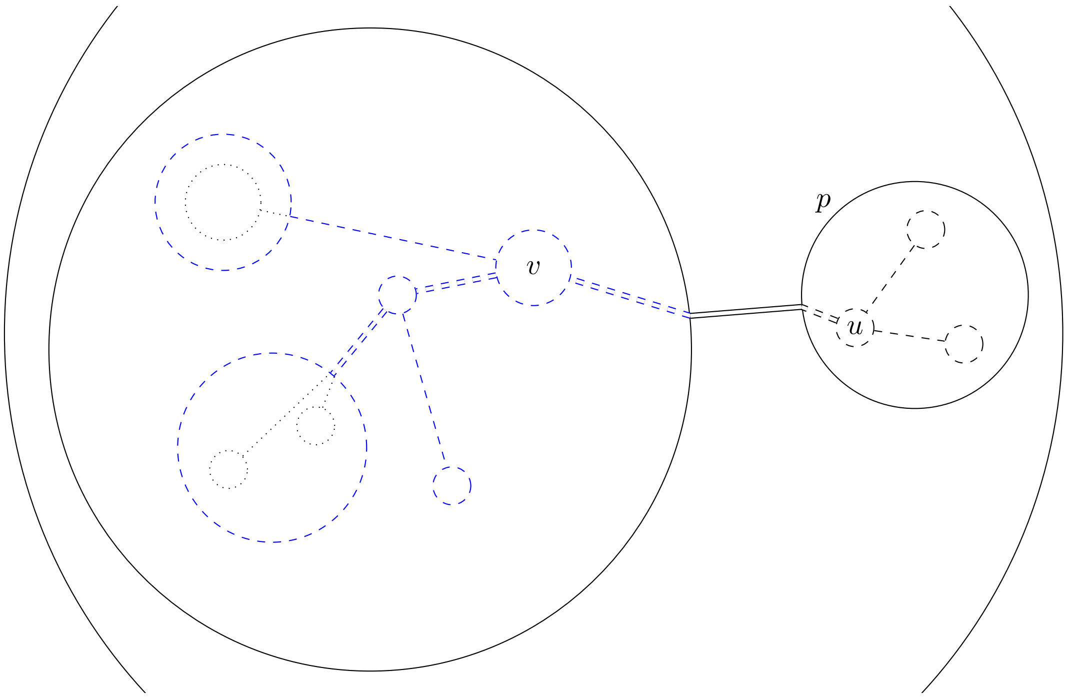


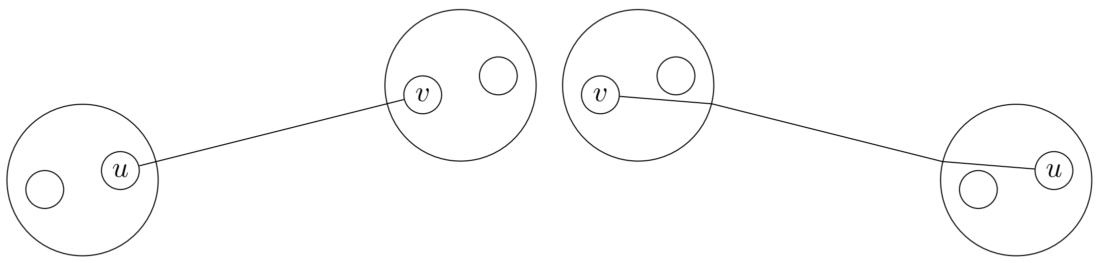
Figure 3.2: Approximation of the position of external vertices.

layout can be constructed. However, even though proper vertex sizes are available to the partial layout algorithm, at the time the partial layout is created the absolute positions of external vertices are unknown, so that external vertices cannot be considered in the computation of the partial layout.

Top-Down Computation

In contrast to the bottom-up order, it is easy to handle external vertices in the top-down order: For the children of the root vertex, the partial layout can be easily computed as there are no external vertices. On a lower level, for a vertex with external edges the top-down order guarantees that all higher levels are completely processed so that the position of either the external vertex or one of its parents – which is not a direct or indirect parent for the original vertex – is already fixed and can be used by the partial layout algorithm to place the vertex. An example is given in Figure 3.2: We want to compute the partial layout highlighted in blue assuming that all vertices with continuous lines have been placed before. Vertex v is adjacent to the vertex u, whose position is not yet known, however, the position of its parent p is determined.

There remains a serious problem with this approach. Because the size of a vertex depends on its not yet computed partial layout, the partial layout algorithm has to predict the children's sizes; this may cause errors such as overlapping vertices along with quality issues due to unused space.



(a) Optimal edge routing. (b) Suboptimal edge routing.

Figure 3.3: Parent layout deteriorating the edge routing.

Mixed Computation

Bottom-up and top-down orders manifest different problems: Whereas the former facilitates the size computation of vertices, the latter allows the inclusion of external vertices in the partial layout computation. It stands to reason to combine the two orders, for example by computing a bottom-up layout at first and refining it in top-down order. Nevertheless, even if large buffers are regarded, the top-down order cannot guarantee a drawing without overlapping vertices as the size of child vertices might increase arbitrarily. Since the bottom-up order does not possess this deficit, one could start with a top-down layout using fixed sizes or intelligent size estimations. However, finishing with a bottom-up layout may render any effort to consider external vertices useless. As shown in Figure 3.3, because the children's layout is computed prior to the parent's layout, parent vertices can strongly alter their positions.

Nonetheless, a mixed approach remains promising. Just as it is used in force-directed algorithms to weaken changes, annealing could be applied to the global layout described here. One could, for example, distribute the computation of the partial layout on several runs of a bottom-up global algorithm. That means that if the partial algorithm runs for 1000 steps, at first 100 steps are executed for each partial layout in bottom-up order, then the next 100 steps and so on until all steps are completed. Of course, it is necessary to decrease the temperature over all 1000 steps. In this way, changes decrease towards the last bottom-up run so that a situation as in Figure 3.3 is less likely to occur. One could also mix runs in bottom-up and top-down orders, but due to the annealing it is probably sufficient to rely only on bottom-up runs. Yet it might be useful to execute the bottom-up run in breadth-first order, as explained for the top-down order, to optimize the quality of the layout.

3.1.4 Simplifications

We neglect some previously mentioned possibilities to simplify the process of creating the software map without losing track of the main requirements. First, we briefly discuss external vertices, but we only compute a simple bottom-up layout and ignore their influence on vertex placement. We also disregard edge weights, and even though combining vertex placement and edge routing might lead to better results, we isolate these steps in our algorithm. At the moment, we cannot imagine a unified approach that is practical. Moreover, we only bundle edges that share a target or origin and leave out edges only related by vicinity. It is possible to bundle edges repeatedly, but we do not use nested bundles. Edge routing could be implemented with line segments or curves; we choose line segments for their low complexity. However, we use Bézier curves to render the drawing in order to give an impression of a smoother layout. Finally, we do not reduce crossings except for edges merged into the same bundle.

3.1.5 Algorithm Sketch

Our algorithm for the complete layout is divided into three major and several smaller steps. First, it is necessary to transform the input multigraph into an ordinary graph by bundling edges. The next step is to construct a collision-free vertex layout by computing the partial layout for each vertex; the vertices are processed in a bottom-up order. The subalgorithm used for creating the partial vertex layout builds a simple layout using an adapted tree-layout algorithm and adjust it with a modified Fruchterman-Reingold algorithm. Afterwards, a heuristic is applied to iteratively route edge bundles while crossings between neighboring bundles are removed via anew bundling. Neighboring bundles are inside the same vertex, target a similar vertex and converge, when they leave a vertex. Finally, the course of each single edge is fixed according to the routing of the bundles and crossings between edges within a bundle are reduced.

1. Graph transformation (including edge bundling)
2. Vertex layout
   - Generate initial partial layout for each vertex
   - Refine partial layout for each vertex
   - Combine partial layouts to global layout
3. Edge layout
   - Route bundles and reduce bundle crossings
   - Route edges and reduce edge crossings

For a better understanding, we define some terms used repeatedly in this thesis before progressing to the description of the algorithm. We already introduced *partial layouts* and *partial graphs* in contrast to the global layout and graph. We also mentioned *internal* and *external* vertices or edges for a specific partial layout. With *parent*, *child* and *root vertex* we refer to the relative position of a vertex in the hierarchy of the software graph. Furthermore, the parent vertex of the vertices in a partial graph is usually denoted as *enclosing vertex.*

3.1.6 Data

The software maps shown in the later parts of the thesis are based on data that stem from several Java projects. We use the Eclipse compiler and the Eclipse Java development tools [22] to analyze the projects and extract the reference graph used as input for our layout algorithm. Nevertheless, these algorithms are neither restricted to Java nor to software graphs; any hierarchic multigraph may be visualized. As for the vertices of the graph, the mapping from code to vertices and edges is done as described in the previous chapter. Still, we limit vertices and edges to code available in text files, along with test code, and neglected code packaged in libraries. In Java, for example, especially regarding smaller projects it would clog the graph if the Java Runtime Library was included. Although it is possible to consider only the parts of libraries that are actually used in a project, these libraries are commonly developed separately from the project, so their relevance is restricted. Of course this decision only concerns the input data; it is not relevant for our layout algorithm and could be changed in another context.

3.2 Graph Transformation

In this section we explain how to transform the hierarchic reference graph into an ordinary graph. Our input is a directed clustered multigraph $G = (V, E, root, H)$ as defined in the preceding chapter, and our objective is to find an undirected graph that bundles the edges of the original graph and hence has no parallel edges. The reason for this is that by computing a layout for the transformed graph, we can also compute a layout for the original graph.

To obtain the bundling, we have to segment the edges as illustrated in Figure 3.4 in order to detect sections of edges that traverse the same vertices. The segmentation is also necessary since edges in our hierarchic reference graph are not local: They may connect any pair of leaf vertices and span several levels of the hierarchy. By segmenting the edges, we can assign the respective parts of the edges, the segments, to the traversed vertices. Moreover, a connection between two vertices also implies a relation between the parents of the vertices, and this information is given by the segmentation.

We denote the transformed graph $SG = (N, S, nv, se)$ as *segment graph* with vertices or *nodes* N and edges or *segments* $S \in N \times N$. Because segments are undirected, a segment $s_1 = (u, v)$ is equivalent to the segment $s_2 = (v, u)$. As illustrated in the figure, each edge is split at vertex borders and parts that have a similar target and traverse the same vertex are assigned to the same segment. In contrast to edges, no segment spans more than one level in the hierarchy. Segments connecting two vertices of the same level are referred to as *root segments*, other than segments between parent vertices and child vertices that are directed towards the border of the parents. Each vertex of the original graph is represented by several nodes, more precisely, by one node for each segment beginning or ending at the respective vertex. These nodes are only moved indirectly when vertices are displaced. They serve as join points for internal segments pointing towards the border of the vertex; edges leaving vertices are split at these points. Node and edge associations are given by the mapping functions $nv : N \to V$

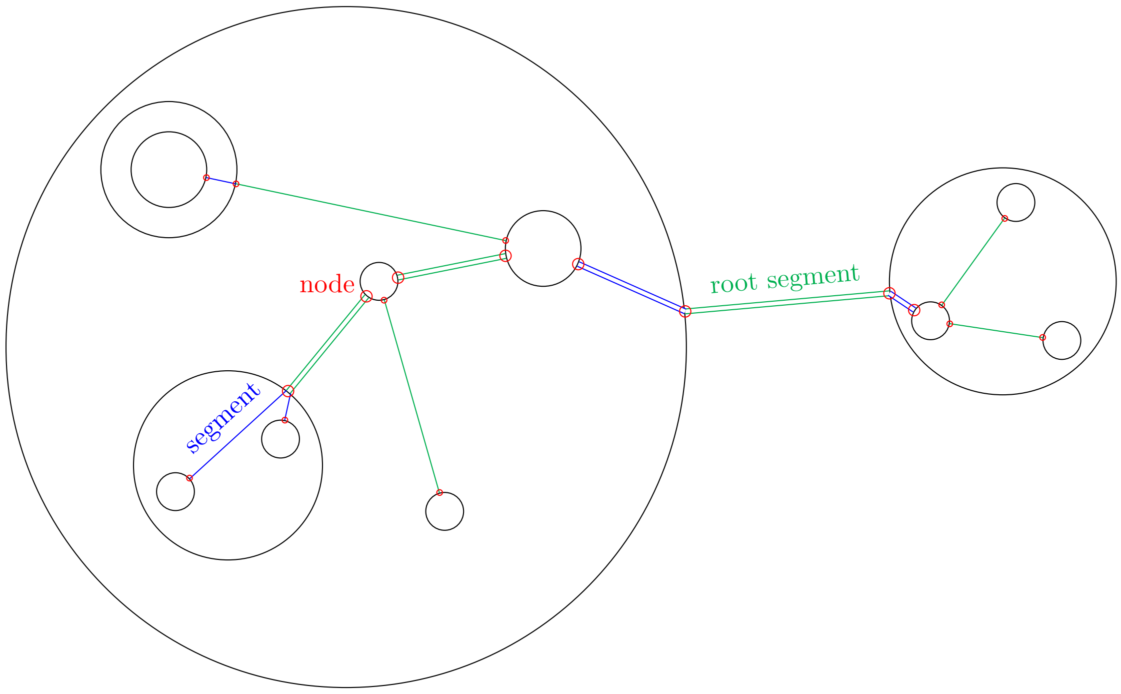


Figure 3.4: Graph with nodes and segmented edges.

and $se : S \to \mathcal{P}(E)$. While segments may bundle several edges, a node is always related to exactly one vertex. We assume that the information given by the original graph, in particular the vertex set V, the edge set E and the hierarchy (r, H), is also incorporated into the segment graph.

Next, we describe how to construct the node and segment sets. As for the segment set, each edge can be converted into a sequence of single-edge segments by using the hierarchy of the graph. For the edge $e = (u_1, v_1)$ displayed in Figure 3.5, let $p_e = \{u_1, u_2, ..., u_i, p, v_j, ..., v_2, v_1\}$ denote the sequence of parents of u_1 and v_1, where the following conditions hold:

$$\forall k \in \{1, ..., i-1\} : (u_k, u_{k+1}) \in H$$
$$(u_i, p) \in H$$
$$(u_j, p) \in H$$
$$\forall k \in \{1, ..., j-1\} : (v_k, v_{k+1}) \in H$$

Next, we explain how to derive the segments for edge e via p_e. We refer only to the segments for vertices u_k as the vertices v_k can be treated accordingly. Since parent vertices always enclose their children, each vertex $u_k \in p_e$ besides u_1 and p indicates that the edge e crosses the border of vertex u_k. Because p encloses all vertices in p_e and u_1 is an end vertex of e, the borders of these vertices are not crossed. Let $n(u_k, e)$ denote the node for vertex u_k to which a segment of e outside of u_k connects as sketched

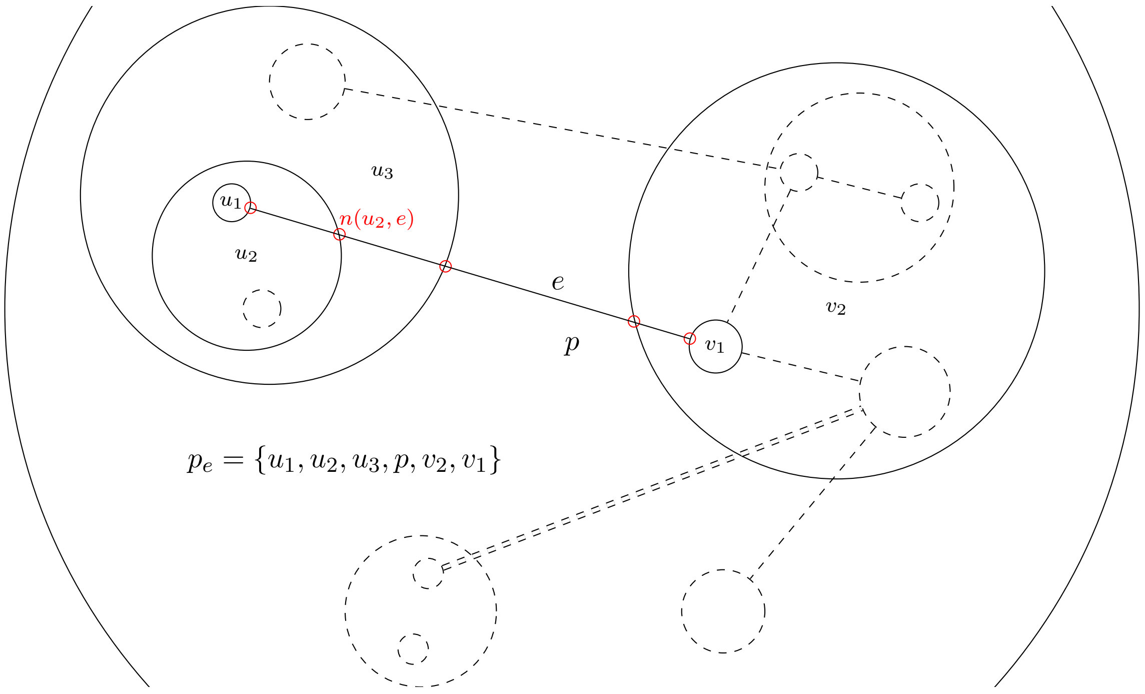


Figure 3.5: Segmentation of a single edge.

in Figure 3.5. For $k > 1$, each node $n(u_k, e)$ is placed on the border of vertex u_k and marks the point at which an inner segment and an outer segment connect. Thus, we can form a sequence of segments $S_{(u,e)} = \{(n(u_k), n(u_{k+1}), \{e\}) : k \in \{1, ..., i\}\}$ for edge e. Finally, we can connect the sequences $S_{(u,e)}$ and $S_{(v,e)}$ to the complete sequence S_e with the root segment $s_r = (n(u_i), n(v_j), \{e\})$. This segment joins the only two vertices sharing the same parent and thus the vertices and the segment are placed inside the parent. Contrary to the other segments, the root segment is also the only segment that does not connect two levels of the hierarchy.

Finally, we can find the common segments for different edges by comparing their parent sequences p_{e_1} and p_{e_2}. As shown in Figure 3.6, two edges e_1 and e_2 share segments if and only if they have the same root segment; we omit proofs at this point. Assuming that the edges e_1 and e_2 possess the same root segment, then their parent sequences p_{e_1} and p_{e_2} both contain a partial sequence $p_{e_1+e_2} = p_{e_1} \cap p_{e_2} = \{u_k, u_{k+1}, ..., u_i, p, v_j, ..., v_{l+1}, v_l\}$ for $k, l \geq 1$. If we derive the respective segment sequences S'_{e_1} and S'_{e_2} for $p_{e_1 \cap e_2}$ as described above, we can merge two segments $s_{e_1} = (n(u_m, e_1), n(u_{m+1}, e_1), \{e_1\})$ and $s_{e_2} = (n(u_m, e_2), n(u_{m+2}, e_2), \{e_2\})$ for $k \leq m \leq (i-1)$ – again accordingly for vertices v_* – in two steps. First, we have to merge the nodes $n(u_m, e_1)$ and $n(u_{m+1}, e_1)$ with their counterparts for e_2. We denote the merged nodes as $n(u_m, e_1 + e_2)$ and $n(u_{m+2}, e_1 + e_2)$. Second, we can merge the segments s_{e_1} and s_{e_2} to the segment $s_{e_1+e_2} = (n(u_m, e_1 + e_2), n(u_{m+2}, e_1 + e_2), \{e_1, e_2\})$.

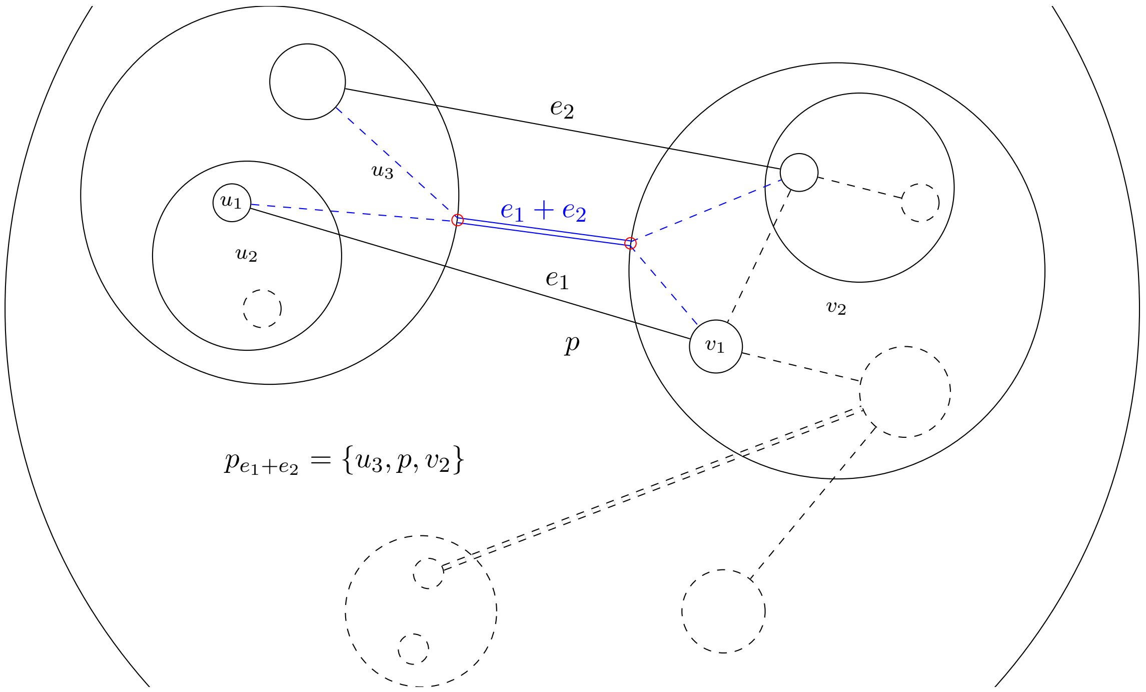


Figure 3.6: Segmentation and bundling of two edges.

The bundling technique presented in this section is similar to the one proposed by Holten [32] as seen in Figure 3.7. Nevertheless, we only use it to segment the graph and convert it to a graph without multiple edges; we use a different method to compute a routing according to the requirements given in the previous chapter.

Now, we can compute the complete segment graph by simply iterating over all edges and repeatedly merging the segments and nodes created by these edges. The algorithm we used is clearly not optimal, but its execution time was still largely dominated by the computation of the layouts, which is why we did not try to find a better or even optimal algorithm.

3.3 Initial Layout

As mentioned in the first section, a force-directed algorithm is used to compute the main layout. If exceptional cases like vertices located on the same position are handled, such an algorithm is able to produce a proper layout starting from any other layout. However, these algorithms do not necessarily perform well for arbitrary initial layouts, so for example random placement of vertices could lead to slower convergence and may degrade the quality of the final layout. In contrast, an initial layout that already reflects the main properties of the underlying graph is more likely to accelerate convergence and generate high-quality layouts [24]. In this work, the following requirements should be met by an initial layout:

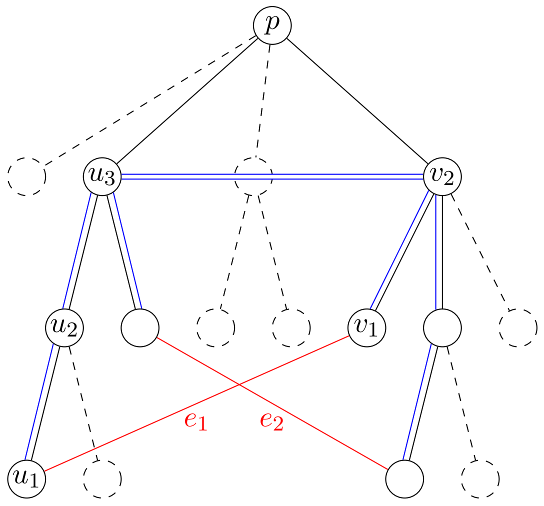


Figure 3.7: Edges (red) and their projections onto the hierarchy (blue).

- **Radii:** The radius of each vertex should be big enough to contain all children.
- **Parent-child-relationships:** All child vertices should be placed inside their parent.
- **Vertex collisions:** No two vertices should overlap.
- **Graph structure:** Thicker segments – segments containing more edges – should cause the connected vertices to be placed near to each other.

While the first three properties are clear, the last criterion is rather vague. As explained later, we intend to fulfill it using minimum spanning trees. Segment collisions, that is segments overlapping with vertices, and segment crossings may occur in the initial layout, and segments connecting external vertices are ignored. The general idea is to compute the global layout bottom-up starting with the leaf vertices; for the partial layout of each vertex a simple radial tree layout is used.

Leaf vertices only need to be assigned a radius r, which could be chosen as a fixed constant R. However, it is not preferable that segments starting in a vertex are wider than the vertex itself, so for setting the radius the number of edges with this vertex as end point should be considered. According to the segment width definition given later, the actual radius of leaf vertices is chosen as the maximum of R and the width of a segment bundling all edges ending at this vertex:

$$r = \max(R, \deg(v) \cdot \mathrm{WIDTH} + (\deg(v) - 1) \cdot \mathrm{SPACE})$$

For each other vertex, the partial layout is generated by computing a tree containing all children of the vertex and computing a radial tree layout. Finally, the radius of the enclosing vertex can be adjusted so that all child vertices are located within the disk defined by the vertex.

By using Kruskal's algorithm [35] we obtain a minimum spanning forest for the local graph. Since we want thicker segments to be included in this forest, we choose the segment weights such that segments with more edges have smaller weight:

$$\text{weight}(s) = \frac{1}{|\text{edges}(s)|}$$

Furthermore, for each tree of the forest, we choose a vertex of the Jordan center [30, p. 35] of the tree to be its root. The Jordan center of a graph $G = (V, E)$ is the set of all vertices with lowest eccentricity. For a vertex u, the eccentricity $e(u)$ is the length of the longest path from the shortest paths to all other vertices $v \in V$. If $\text{path}(u, v)$ denotes the shortest path between two vertices u and v, then the eccentricity can be defined as follows:

$$e(u) = \max_{v \in V}(|\text{path}(u, v)|)$$

So a Jordan center is a vertex with the smallest distance to all other vertices in the graph. In the end, one of these roots is placed in the center of the enclosing vertex, for which reason it is desirable to choose such a vertex. However, for computing the Jordan center we need a means to compute the distance between connected vertices. We use a function $d_{\text{pref}} : V \times V \to \mathbb{R}^{>0}$ to map two vertices onto their *preferred distance*. This function is defined in detail in a later section; for now, we simply assume that two vertices u and v can not collide if their distance is greater than or equal to $d_{\text{pref}}(u, v)$.

Although the Jordan Center could already be computed, a slight redefinition of the shortest path between two vertices u and v is beneficial. As can be seen in Figure 3.8, the Jordan center is the vertex v between u and w. But even though the eccentricity of the vertex v is minimal, the enclosing disk centered at u would be much smaller than that centered at v due to the larger radius of u. By simply adding the radius of the target vertex to the length of the path we can change the Jordan center to u so that it reflects the actual distance to the border of the graph:

$$e'(u) = \max_{v \in V}(|\text{path}(u, v)| + r_v)$$

It should be mentioned that this function is no longer symmetric; $\text{path}(u, v)$ might yield a different result than $\text{path}(v, u)$. Finally, to find a Jordan center vertex in the tree we compute the eccentricity for each vertex by enumerating the distances to all other vertices and select a vertex with minimal eccentricity.

In a third step, the forest is merged by choosing one tree and adding the roots of all other trees as children to its root. Here it is important to note that for the rest of this section, we do not indicate the hierarchy of the complete graph by using the terms parent or child, but instead vertices with respect to their position in the tree. The main tree could be selected arbitrarily; however, since its root is centered in the end, it should be advantageous if the largest tree is chosen. Hence, we define a heuristic size as follows and select a tree t' with maximum size.

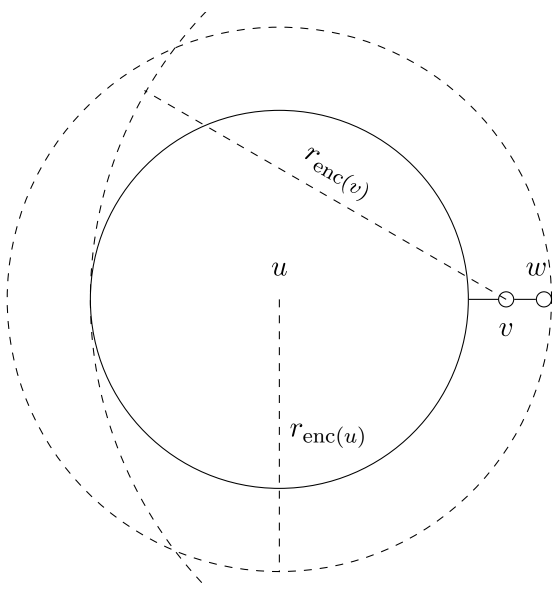


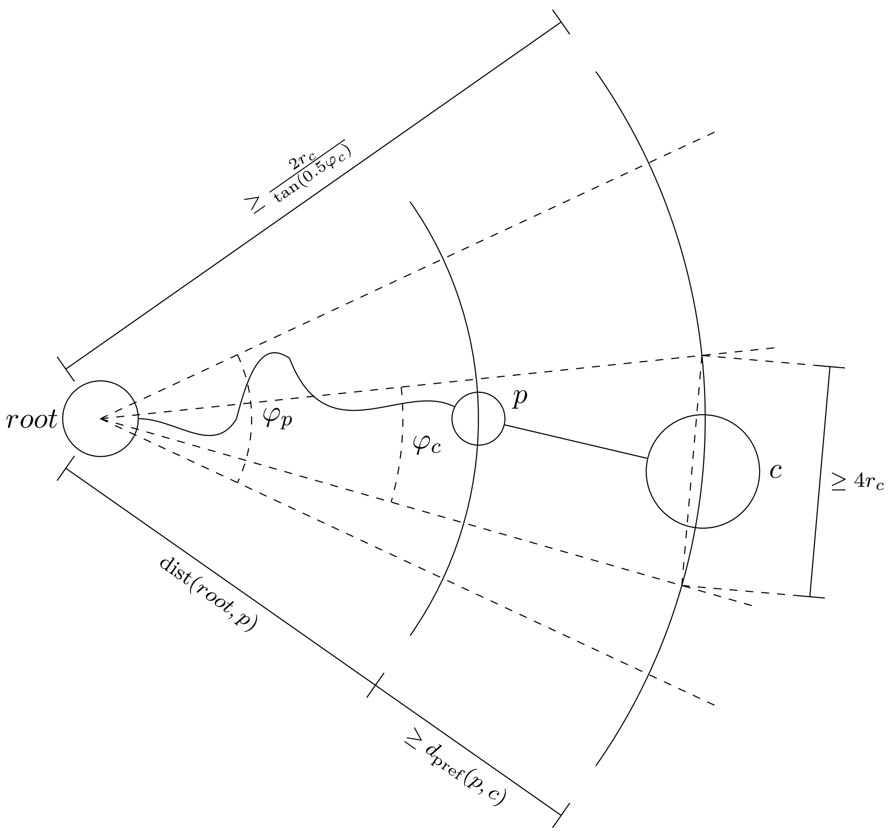
Figure 3.8: Selecting the Jordan center.

$$\text{size}(T) = \sum_{\forall v \in T} r_v$$

Finally, we can compute the partial layout for the enclosing vertex by applying a radial tree layout, in our case a modified level-based algorithm [18]. Similar to known approaches, we distribute the child vertices radially around the root by assigning an angle to each child and placing the subtree rooted at it in the area determined by the angle. Commonly, the angle is chosen according to the number of vertices in the subtree, but since we use two-dimensional vertices, we apply the heuristic size definition given above. If T_v denotes the subtree rooted in a vertex v, then for a parent vertex p with an assigned angle of φ_p, the angle assigned to a child c of p is defined as follows:

$$\varphi_c = \varphi_p \cdot \frac{\text{size}(T_c)}{\text{size}(T_p)}$$

For obvious reasons, the angle for the root vertex is 2π. In the original algorithm, the angle for children may be restricted further to prevent crossings in the tree layout. Nevertheless, our initial layout is not the final layout; actually it is not a pure tree layout and the inclusion of the remaining edges might cause other crossings so that the effort put into such a condition is wasted. In contrast, we have to deal with another problem concerning vertex collisions: Because we use two-dimensional vertices instead of zero-sized points, it is not possible to place the child vertices at arbitrary distances from their parents or they may overlap. Moreover, we have to avoid collisions between children of different parents by placing them inside of the wedge defined by the respective parents'



$\geq \frac{2r_c}{\tan(0.5\varphi_c)}$
$root$
φ_p
φ_c
p
c
$\geq 4r_c$
dist($root, p$)
$\geq d_{\text{pref}}(p, c)$

Figure 3.9: Space estimation for child vertices.

angle. However, let φ_c denote the angle for a child c with radius r_c, then the following conditions for the distance between c and its parent p allow for a collision-free layout:

$$\text{dist}(p, c) \geq d_{\text{pref}}(p, c)$$
$$(\varphi_c < \pi) \rightarrow \text{dist}(root, c) \geq \frac{2r_c}{\tan(0.5\varphi_c)}$$

We assumed that two vertices can not overlap if their distance is greater or equal to their preferred distance, so p and c may never overlap. If $\varphi_c \geq \pi$ holds, the angle for c spans more than half a circle, the subtree T_c is completely enclosed in the wedge with angle φ_c and collisions between vertices outside of the subtree T_c cannot occur. Otherwise, we can assume a situation as illustrated in Figure 3.9. Clearly, the base of the shown triangle is longer than $4r_c$ if we fulfill the constraints defined before, and hence, the wedge contains the child c completely. Consequently, we can avoid any collision if we choose the distance as follows:

$$\operatorname{dist}(root, c) = \begin{cases} \operatorname{dist}(root, p) + d_{\text{pref}}(p, c) & \text{if } \varphi_c \geq \pi \\ \max\left(\operatorname{dist}(root, p) + d_{\text{pref}}(p, c), \frac{2r_c}{\tan(0.5\varphi_c)}\right) & \text{if } \varphi_c < \pi \end{cases}$$

Finally, Algorithm 3.1 computes the radial tree layout for a given tree. The root vertex is placed in the center of the enclosing vertex and the position of a child vertex c is computed relative to the enclosing vertex by rotating the vector $\vec{e}$ based on the angles of the respective wedge; φ_{start} denotes the angle at which the wedge begins. $\vec{e}$ is an arbitrary, but fixed unit vector. We can obtain the correct length by multiplying it with the distance between c and the root vertex.

After computing the layout, we have to adjust the size of the enclosing vertex so that its partial layout fits inside. Since this step is also necessary in the force-directed layout, we describe it in detail in the next chapter.

Algorithm 3.1 Radial tree layout

```
procedure LAYOUT($root$)
    $\text{pos}(root) \leftarrow (0, 0)$
    LAYOUT($root$, $root$, 0, $2\pi$)

procedure LAYOUT($root$, $v$, $\varphi_{\text{start}}$, $a$)
    if $v$ is leaf then
        return
    $s_v \leftarrow \text{size}(T_v) - r_v$
    for each child $c$ of $v$ do
        $\varphi_c \leftarrow a \cdot \frac{\text{size}(T_c)}{s_v}$
        $d \leftarrow$ DISTANCE($root, v, c, a$)
        $\text{pos}(c) \leftarrow \text{rotate}\left(\vec{e} \cdot d, \varphi_{\text{start}} + \frac{\varphi_c}{2}\right)$
        LAYOUT($root$, $c$, $\varphi_{\text{start}}$, $\varphi_c$)
        $\varphi_{\text{start}} \leftarrow \varphi_{\text{start}} + \varphi_c$

function DISTANCE($root$, $v$, $c$, $a$)
    $d \leftarrow \text{dist}(root, v) + d_{\text{pref}}(v, c)$
    if $a \geq \pi$ then
        return $d$
    return $\max\left(d, \frac{2r_c}{\tan(0.5\varphi_c)}\right)$
```

3.4 Vertex Layout

In this section, we describe how to refine the vertex placement given by the initial layout. As for the initial layout, we only deal with the positioning of vertices, the edge layout is covered in the next section. First, we explain how to compute the global layout assuming that we are able to compute the partial layout for each vertex, and in the second part we detail the algorithm constructing the partial layout.

3.4.1 Global Layout Algorithm

As described in the first section, we construct the complete layout by traversing the hierarchy, creating the partial layouts and assembling them; the corresponding code is displayed in Algorithm 3.2. We assume that the original graph $G = (V, E, root, H)$ as well as the transformed graph $SG = (N, S, nv, se)$ are accessible in the algorithm without being explicitly specified as parameters; the initial call to compute the layout for graph G is Layout($root$).

Algorithm 3.2 Global layout algorithm

```
1: procedure Layout($v$)
2:     if $c$ is leaf then
3:         return
4:     for each child $c$ of $v$ do
5:         Layout($c$)
6:     PartialLayout($v$)
7:     Resize($v$)
```

Because we do not consider external vertices, it is sufficient to traverse the hierarchy H in a deep-first bottom-up order. Obviously, leaf vertices do not possess a partial layout and since we have already assigned a proper size to them in the initial layout, we do not need to process them again. However, we have to adjust the size of a vertex v after we applied the partial layout algorithm as the space needed by its children may have increased or reduced.

Algorithm 3.3 describes the method we currently use to resize vertices. Supposing that the child vertices are positioned relative to the position of the enclosing vertex v, a disk with radius r_v centered at the origin of v encloses a child c if and only if $r_v \geq |\text{pos}(c)| + r_c$ as seen in Figure 3.10. Accordingly, we can compute the minimum radius for the enclosing disk by computing the maximum of $|\text{pos}(c)|+r_c$ over all children. Moreover, we add a buffer $b \geq 1$ to the radius to prevent that the border of a child coincides with the border of v.

Algorithm 3.3 Resizing of vertices

```
1: procedure Resize($v$)
2:     $r \leftarrow 0$
3:     for each child $c$ of $v$ do
4:         if $r < |\text{pos}(c)| + r_c$ then
5:             $r \leftarrow |\text{pos}(c)| + r_c$
6:     $r_v \leftarrow r$
```

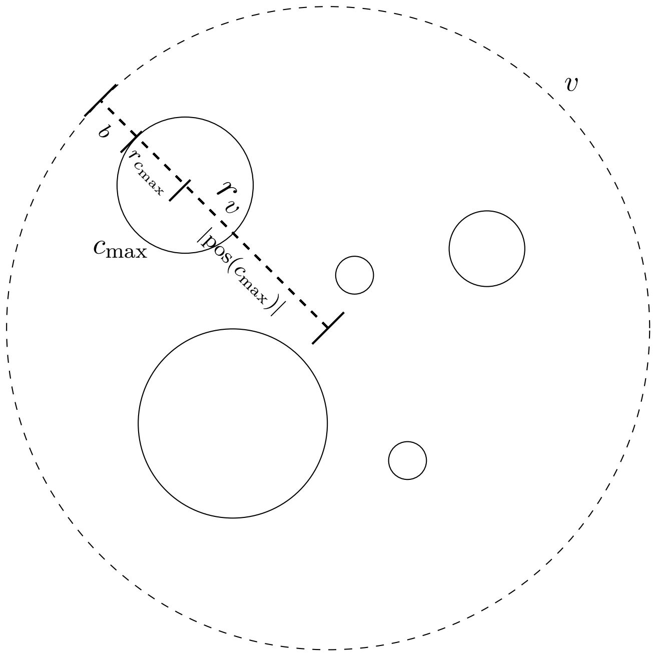


Figure 3.10: Preferred radius for a parent vertex.

3.4.2 Partial Layout Algorithm

In this section, we describe the force-directed algorithm that is responsible for the vertex placement. We employ the method of Fruchterman and Reingold [23] and adapt its force definitions to our purposes. Algorithm 3.4 shows the original layout procedure.

The algorithm computes the layout with respect to a frame given by width W and length L. Furthermore, it tries to distribute the zero-sized points representing the vertices evenly on the area; the variable k denotes the optimal distance for each pair of vertices and is used for determining the repulsive and attractive forces. These forces are defined such that repulsion and attraction are balanced for a connected pair of vertices if their distance is k. To ensure that vertices are not placed next to each other, f_{rep} goes to infinity as the distance shrinks to zero. Similarly, the quadratic function f_{attr} prevents adjacent vertices from being pulled apart too far.

In the main loop, the forces are computed for each edge and vertex and the vertices are moved accordingly. In each iteration, the temperature t is decreased to manipulate the vertex movements: Initially, vertices may be displaced by a greater distance to avoid local minima, whereas in the end vertices are only shifted slightly to improve the layout locally. Often, a linear function is applied to compute the annealing.

While this algorithm performs well for simple graphs, in its present form it is not applicable to our partial graph, so we detail the requirements of the partial layout hereafter.

Algorithm 3.4 Fruchterman-Reingold [23]

```
 1: procedure LAYOUT($G = (V, E)$)
 2:     $area \leftarrow W \cdot L$                              ▷ frame: width $W$ and length $L$
 3:     initialize $G$                                          ▷ place vertices at random
 4:     initialize $t$                                          ▷ initialize temperature $t$
 5:     $k \leftarrow \sqrt{\frac{area}{|V|}}$                  ▷ compute optimal pairwise distance
 6:     for $i \leftarrow 1$ to $iterations$ do
 7:         for each $v \in V$ do                               ▷ compute repulsive forces
 8:             $\text{disp}(v) \leftarrow 0$                   ▷ initialize displacement vector
 9:             for each $u \in V$ do
10:                 if $u \neq v$ then
11:                     $\Delta \leftarrow \text{pos}(v) - \text{pos}(u)$
12:                     $\text{disp}(v) \leftarrow \text{disp}(v) + \frac{\Delta}{|\Delta|} \cdot f_{\text{REP}}(|\Delta|)$
13:         for each $(v, u) \in E$ do                          ▷ compute attractive forces
14:             $\Delta \leftarrow \text{pos}(v) - \text{pos}(u)$
15:             $\text{disp}(v) \leftarrow \text{disp}(v) - \frac{\Delta}{|\Delta|} \cdot f_{\text{ATTR}}(|\Delta|)$
16:             $\text{disp}(u) \leftarrow \text{disp}(u) + \frac{\Delta}{|\Delta|} \cdot f_{\text{ATTR}}(|\Delta|)$
17:         for each $v \in V$ do                               ▷ displace vertices within frame
18:             $\text{pos}(v) \leftarrow \text{pos}(v) + \frac{\text{disp}(v)}{|\text{disp}(v)|} \cdot \min(|\text{disp}(v)|, t)$
19:             $\text{pos}(v).x \leftarrow \min\left(\frac{W}{2}, \max\left(-\frac{W}{2}, \text{pos}(v).x\right)\right)$
20:             $\text{pos}(v).y \leftarrow \min\left(\frac{L}{2}, \max\left(-\frac{L}{2}, \text{pos}(v).y\right)\right)$
21:         $t \leftarrow \text{cool}(t)$
22:
23: function $f_{\text{REP}}(distance)$
24:     return $\frac{k^2}{distance}$
25:
26: function $f_{\text{ATTR}}(distance)$
27:     return $\frac{distance^2}{k}$
```

Problem Definition

Let $SG_w = (N_w, S_w, nv, se)$ denote the possibly disconnected partial segment graph for a vertex w, with the respective subsets only containing the relevant nodes and segments and V_w being the set of vertices that are children of w. Our input for the partial layout algorithm is SG_w plus the positions and sizes for all vertices in V_w as determined by the initial layout. As before, positions are considered relative to the origin of w, not globally. For the vertex placement, we want to meet the following requirements:

- **Vertex collisions:** No two vertices should overlap.
- **Graph structure:** While adjacent vertices should be placed near to each other, unrelated vertices should disperse.
- **Vertex distance:** The distance between vertices should grow with their size, moreover, there should be enough space for the later edge routing.
- **Layout compaction:** No vertex should be placed too far from the other vertices of the partial graph.
- **Circular layout:** Since the shape of the parent vertex is a disk, the partial layout should approximate a disk as well.

In comparison to the characteristics of the initial layout, most modifications and additions should be intuitive. The third point is especially important because we separate vertex and edge layout: If we lead edges around vertices without moving vertices, we have to ensure that there is enough space between vertices. The algorithm given before does indeed respect the graph structure, that is, it positions connected vertices closely and pulls disconnected ones apart. Edges are regarded as unweighted, but this does not pose a problem since we also neglect edge weight as stated in the beginning of this chapter. However, there are several flaws in the algorithm and it is not capable of producing a layout complying with our demands.

The first issue is that the algorithm of Fruchterman and Reingold is designed for zero-sized vertices. Taking two-dimensional vertices into account, the algorithm does not keep vertices from overlapping, which makes it necessary to redefine repulsion and attraction. By redefining these forces we can also change the optimal distance according to the radii of vertices. Furthermore, we choose a minimum size for leaf vertices and due to nesting, the size of parent vertices may grow arbitrarily, so that it is difficult to predict the size required for the layout of a graph. On this account, we do not dictate a frame size and treat the layout as unbounded, which again influences repulsion and attraction. Lacking a frame, we also have to reconsider the temperature: Concerning partial graphs with larger vertices, we have to allow for much stronger movements to overcome local minima; if the temperature is too low, it might even happen that overlapping vertices are moving too slowly to escape each other. Finally, the input graph may be disconnected, and in the given algorithm unrelated vertices are placed on the border of the frame. Even more, since we do not intend to bound the layout a drawing with disconnected subgraphs would fall apart.

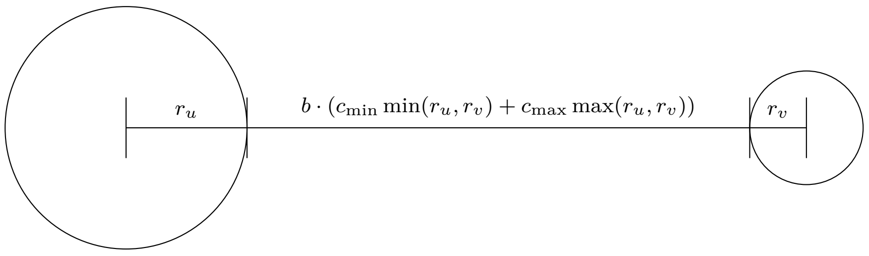

r_u
$b \cdot (c_{\min} \min(r_u, r_v) + c_{\max} \max(r_u, r_v))$
r_v

Figure 3.11: Preferred distance between two vertices.

Preferred Distance

Prior to moving on to forces, we specify the already mentioned preferred distance between two vertices; the redefined forces strongly rely on it. In the standard algorithm of Fruchterman and Reingold, the parameter k is computed using the frame size and denotes the optimal pairwise distance with respect to an even vertex distribution. Instead, we define the function for the preferred distance of two vertices u and v as below:

$$d_{\text{pref}}(u, v) = r_u + r_v + b \cdot (c_{\min} \min(r_u, r_v) + c_{\max} \max(r_u, r_v)), b \geq 1, c_{\min}, c_{\max} > 0$$

The addition of the radii r_u and r_v ensures that vertices do not overlap if they conform to the preferred distance. Furthermore, such vertices do not touch each other due to the positive summand $b \cdot (c_{\min} \min(r_u, r_v) + c_{\max} \max(r_u, r_v))$ as seen in Figure 3.11. The parameters $c_{\min}$ and $c_{\max}$ are handles for controlling how the radii of the vertices influence the preferred distance. A simpler option would be to use the average radius for computing the preferred distance, however, it is helpful to amplify the influence of the smaller radius. We discuss how to choose the parameters and their effect in detail in the next chapter. The last parameter, b, is a means to regulate the distance between vertices in general; we use it for adjusting the forces in the following section. By adjusting these paramters, we can achieve an appealing vertex layout with enough space for routing the edges.

Force Definition

Let $d_{\text{actual}}(u, v) = |\text{pos}(u) - \text{pos}(v)|$ denote the distance for two vertices u and v. In the original algorithm, repulsion and attraction are in equilibrium for two adjacent vertices u and v if $d_{\text{actual}}(u, v) = k$. The easiest way to adjust this equilibrium to the preferred distance defined above is to just replace k in the original force equations by $d_{\text{pref}}(u, v)$:

$$f_{\text{rep}}(u, v) = \frac{d_{\text{pref}}(u, v)^2}{d_{\text{actual}}(u, v)}$$
$$f_{\text{attr}}(u, v) = \frac{d_{\text{actual}}(u, v)^2}{d_{\text{pref}}(u, v)}$$

Now, $f_{\text{rep}}(u,v) = f_{\text{attr}}(u,v)$ holds as wished for $d_{\text{pref}}(u,v) = d_{\text{actual}}(u,v)$ so that u and v are positioned in their preferred distance – provided that there are no other forces interfering. The problem is that for example in a complete graph with more vertices additional attractive forces pull the vertices together, and since the repulsive force given above still only goes to infinity if the distance shrinks to zero, the vertices may happen to overlap. For this reason, we redefine the forces such that while the equilibrium distance is unchanged, the repulsion grows infinitely as soon as the distance shrinks to a fixed minimum distance. We derived this distance directly from the given function $d_{\text{min}}(u,v)$ with $c_{\text{min}} \geq 0$, $c_{\text{max}} \geq 0$ and $c_{\text{min}} + c_{\text{max}} = 1$:

$$d_{\text{min}}(u,v) = r_u + r_v + b_{\text{min}} \cdot (c_{\text{min}} \min(r_u, r_v) + c_{\text{max}} \max(r_u, r_v)), b_{\text{min}} \geq 1, c_{\text{min}}$$
$$d_{\text{pref}}(u,v) = r_u + r_v + b_{\text{pref}} \cdot (c_{\text{min}} \min(r_u, r_v) + c_{\text{max}} \max(r_u, r_v)), b_{\text{pref}} > b_{\text{min}}$$

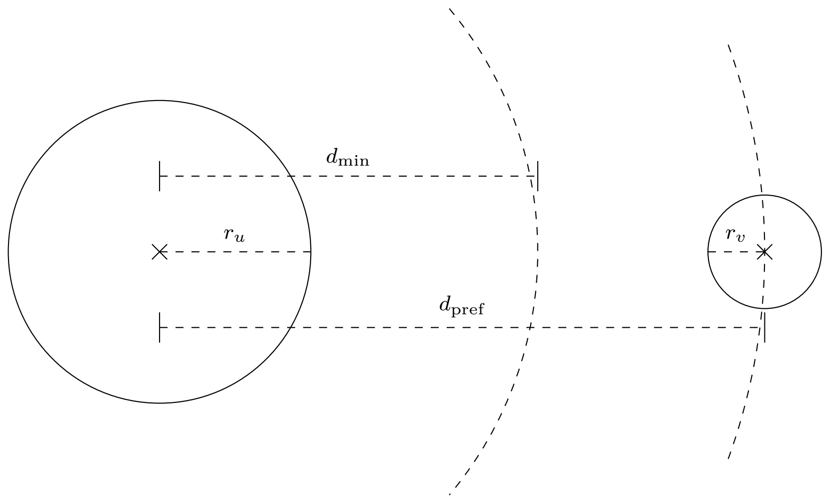
Using the functions $d_{\text{min}}(u,v)$ and $d_{\text{pref}}(u,v)$, we then can redefine the repulsive and attractive forces as follows:

$$d'_X(r_u, r_v) = d_X(r_u, r_v) - d_{\text{min}}(r_u, r_v), X \in \{\text{pref}, \text{actual}\}$$
$$f_{\text{rep}}(r_u, r_v) = \begin{cases} \frac{d'_{\text{pref}}(r_u,r_v)^2}{d'_{\text{actual}}(r_u,r_v)} & \text{if } 0 \leq d'_{\text{actual}}(r_u, r_v) \\ \infty & \text{otherwise} \end{cases}$$
$$f_{\text{attr}}(r_u, r_v) = \begin{cases} \frac{d'_{\text{actual}}(r_u,r_v)^2}{d'_{\text{pref}}(r_u,r_v)} & \text{if } 0 \leq d'_{\text{actual}}(r_u, r_v) \\ 0 & \text{otherwise} \end{cases}$$

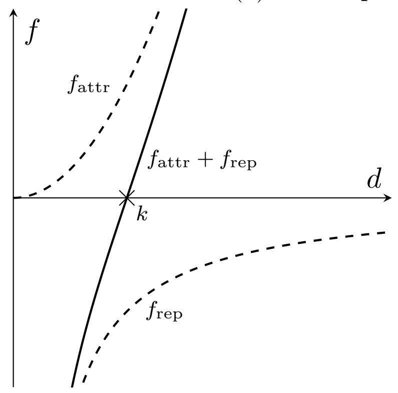
Here, $d'_X(r_u, r_v)$ serves as simplification for the difference between preferred and minimum distance. These forces behave almost as in the original algorithm, the only difference is that they ensure a minimum distance between every vertex pair as illustrated in Figure 3.12. Although this minimum does not guarantee enough space for routing the edges, it was sufficient for computing good layouts in practice. Later, we give a parameter configuration that we successfully used to generate a drawing for real-world graphs.

So far, we considered the first three of our five requirements. We still have to deal with disconnected graphs and a circular-shaped layout. While the first criterion is really important because it prevents the layout from falling apart, the latter is rather a cosmetic issue. We can treat both issues by introducing a third, gravitational force as done in other algorithms. Possible choices are forces that are linear or quadratic with respect to the distance to the origin of the enclosing vertex, which leads us to the more general definition given below:

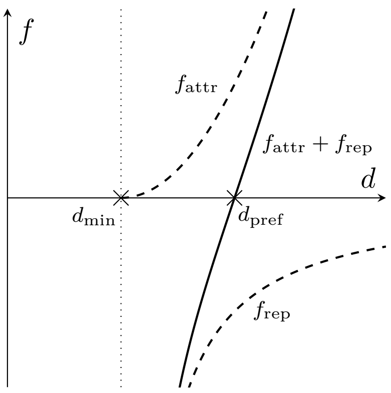
$$f_{\text{grav}}(v) = g \cdot |\text{pos}(v)|^p, g > 0, p \geq 0$$



(a) Vertex pair placed in optimal distance.



(b) Original force function.



(c) Redefined force function.

Figure 3.12: Vertex distance and course of the force function.

Even though this kind of gravitational force does not necessarily produce optimal results, it is simple and scales properly for larger graphs; we discuss its characteristics in the next chapter.

Temperature and Vertex Displacement

In the original algorithm given by Fruchterman and Reingold, the temperature is an absolute boundary that is equal for every vertex. This is sufficient if vertices are represented by points and the layout area is fixed, but in our case vertices are represented by disks and we want to compute a proper layout without dictating the area. For this reason, we parametrize the temperature by the current radius of the enclosing vertex, which can be computed in each iteration as in the procedure Resize((v)). Let par(v) denote the parent of a vertex v, then we can update the position for c as follows:

$$\mathrm{pos}(v) \leftarrow \mathrm{pos}(v) + \frac{\mathrm{disp}(v)}{|\mathrm{disp}(v)|} \cdot \min(|\mathrm{disp}(v)|, t \cdot r_{\mathrm{par}(v)})$$

Other than in the original algorithm, the maximum displacement depends on the current layout and the temperature t only scales the movements. This also means that even if the temperature is a strictly monotonic decreasing function, the step width might increase due to changes in the layout.

Refined Algorithm

Even though the input for our partial layout algorithm is the segment graph SG, we defined forces and other equations with respect to vertices, not nodes, since we only deal with root segments and move vertices. The vertex associated with a node n is denoted as ver(n). Due to our initial layout, we do not need to initialize vertices as in the original algorithm, and we neglect segments other than root segments. Finally, the modifications for the algorithm of Fruchterman and Reingold, which we described above, lead us to Algorithm 3.5.

3.5 Edge Layout

In the next section, we describe how to compute the edge layout after the vertices have been placed. We first define the input and the goals we want to achieve. Next, we refer to solutions for similar problems found by other researchers and discuss their applicability. Finally, we sketch our solution, detail the individual steps and specify an algorithm that solves the given problem.

3.5.1 Problem Definition

At this point, we start with a layout that already meets most of the requirements given in the first chapter: Vertices are represented by disks, they do not overlap, the hierarchy

Algorithm 3.5 Partial layout algorithm

```
 1: procedure LAYOUT($SG_w = (N_w, S_w, nv, se),\ V_w$)
 2:     initialize $t$                                          ▷ initialize temperature $t$
 3:     for $i \leftarrow 1$ to $iterations$ do
 4:         for each $v \in V_w$ do                  ▷ compute repulsive and gravitational forces
 5:             $\text{disp}(v) \leftarrow 0$                          ▷ initialize displacement vector
 6:             for each $u \in V_w$ do
 7:                 if $u \neq v$ then
 8:                     $\Delta \leftarrow \text{pos}(v) - \text{pos}(u)$
 9:                     $\text{disp}(v) \leftarrow \text{disp}(v) + \frac{\Delta}{|\Delta|} \cdot f_{\text{REP}}(u,\ v)$
10:             $\text{disp}(v) \leftarrow \text{disp}(v) - \frac{\text{pos}(v)}{|\text{pos}(v)|} \cdot f_{\text{GRAV}}(v)$
11:         for each $(n_v, n_u) \in S_w$ do                      ▷ compute attractive forces
12:             if $(n_v, n_u)$ is root segment then
13:                 $v \leftarrow \text{ver}(n_v)$
14:                 $u \leftarrow \text{ver}(n_u)$
15:                 $\Delta \leftarrow \text{pos}(v) - \text{pos}(u)$
16:                 $\text{disp}(v) \leftarrow \text{disp}(v) - \frac{\Delta}{|\Delta|} \cdot f_{\text{ATTR}}(u,\ v)$
17:                 $\text{disp}(u) \leftarrow \text{disp}(u) + \frac{\Delta}{|\Delta|} \cdot f_{\text{ATTR}}(u,\ v)$
18:         for each $v \in V$ do                                          ▷ displace vertices
19:             $\text{pos}(v) \leftarrow \text{pos}(v) + \frac{\text{disp}(v)}{|\text{disp}(v)|} \cdot \min(|\text{disp}(v)|, t \cdot r_w)$
20:         $t \leftarrow \text{cool}(t)$
21:
22: function $f_{\text{REP}}(u,\ v)$
23:     if $0 > d'_{\text{actual}}(r_u, r_v)$ then
24:         return $\infty$
25:     return $\frac{d'_{\text{pref}}(r_u,r_v)^2}{d'_{\text{actual}}(r_u,r_v)}$
26:
27: function $f_{\text{ATTR}}(u,\ v)$
28:     if $0 > d'_{\text{actual}}(r_u, r_v)$ then
29:         return 0
30:     return $\frac{d'_{\text{actual}}(r_u,r_v)^2}{d'_{\text{pref}}(r_u,r_v)}$
31:
32: function $f_{\text{GRAV}}(v)$
33:     return $g \cdot |\text{pos}(v)|^p, g > 0, p \geq 0$
```

is respected and related vertices are placed near to each other. To complete the layout, we have to fulfill the four remaining constraints listed below.

- **Edge shapes:** Edges are represented by a series of lines or curves.
- **Collision-free layout:** No vertex v overlaps with or touches an edge not incident to v.
- **Edge bundling:** Edges with a similar origin or target should be routed together.
- **Edge routing:** Edges should not deviate strongly from their ideal route and unnecessary crossings should be avoided.

While the edge bundling is nearly covered by the transformation to the segment graph, we still have to draw the single edges according to this bundling. Besides that, the main point is to introduce a suitable, non-overlapping polyline for each segment.

3.5.2 Known Approaches

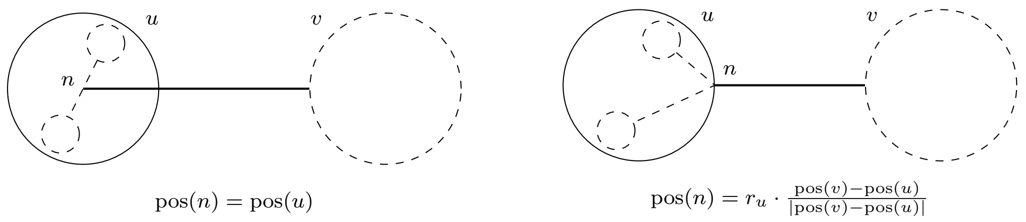
We already referred to the hierarchic bundling introduced by Holten [32], which is similar to our graph transformation. In addition to this technique, in a joint work with van Wijk [33] Holten presented another method to bundle edges of non-hierarchic graphs; they use forces to pull edges together locally. Gansner et al. [25] showed a different approach for bundling edges according to a proximity metric and Ersoy et al. [20] find common routes for edges by carving a skeleton out of a graph. Moreover, Pupyrev et al. [40] introduced a technique for both routing and bundling edges by minimizing path length.

Respective to edge routing, an early work is the paper by Dobkin et al. [14] in which edges are represented by obstacle-avoiding paths consisting of splines. More recent approaches were presented by Dwyer et al. [16]: A first work deals with the embedding of edge routing in a force-directed algorithm; it also tries to minimize crossings. Later, Dwyer [17] contributed to a second paper in which simplified visibility graphs are used to route edges of large graphs.

Besides the first mentioned work by Holten, all techniques refer to non-hierarchic graphs and could be used for computing the partial layout. However, for simplicity we mostly ignore edge bundling in the partial graphs apart from the already described hierarchic bundling. Furthermore, we show an alternative method for routing edges by employing an easy geometric heuristic.

3.5.3 Algorithm Sketch

We start with a layout in which each segment is represented by a straight line between its end nodes, whose thickness is determined by the amount of edges associated with the respective segment. The basic idea is to incrementally modify this graph by checking segments for collisions with vertices and inserting new nodes avoiding these collisions until all collisions are removed. Besides, we remove crossings of segments that were



u
v
n
$\mathrm{pos}(n) = \mathrm{pos}(u)$
u
v
n
$\mathrm{pos}(n) = r_u \cdot \frac{\mathrm{pos}(v)-\mathrm{pos}(u)}{|\mathrm{pos}(v)-\mathrm{pos}(u)|}$

(a) Node centered.

(b) Node placed on border.

Figure 3.13: Node positions for root segments.

introduced by inserting nodes. Finally, we convert the routing of the segments into a corresponding edge routing that is – assuming that the segment routing is collision-free – collision free as well. In this work, we limit ourselves to computing a routing consisting of a series of lines instead of curves.

3.5.4 Bundle Routing

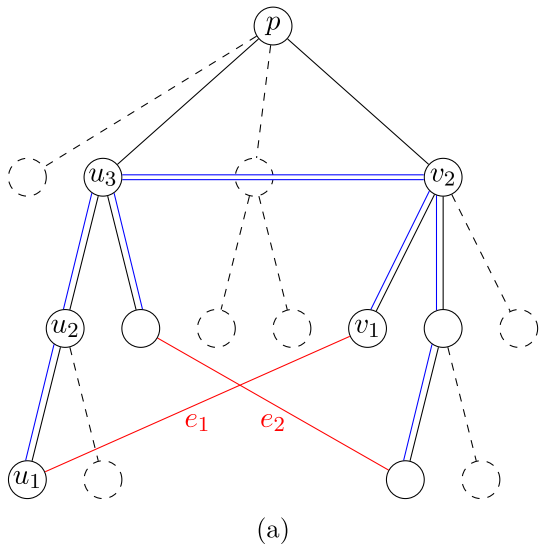
We already computed the positions and radii for all vertices. To obtain a straight-line drawing with segments, we only have to define the positions of their end nodes and connect them with a thick line. For root segments, we could simply use the centers of the involved vertices as node positions. However, as seen in Figure 3.13a this would not lead to a satisfying result considering the connections to child vertices. Instead, we place the nodes at the border of the respective vertex; this position can easily be computed as shown in Figure 3.13b.

If we go back to the projection of segments onto the hierarchy, then it is easy to see that the end nodes of a root segment can be regarded as roots of two trees consisting of the segments ending at the root segment as depicted in Figure 3.14. For a non-root segment $s = (n_u, n_v)$ connecting the vertices u and v, we then can compute the node positions of n_u and n_v as sketched in Figure 3.15. It is important to mention that $s = (n_u, n_v)$ is not necessarily the only segment connecting the vertices u and v as indicated in the figure by the nodes n'_u and n'_v; this condition only holds for root segments.

Based on that, we can compute all node positions recursively starting from the root segments' positions. Let $s = (n_u, n_v)$ denote a segment that is either a root segment or, without loss of generality, for which v is the unique parent of u. The position of the node n_u is then given by the formula below; in contrast to previous definitions, henceforth we use absolute positions abs$(*)$ for vertices and nodes.

$$\mathrm{abs}(n_u) = \mathrm{abs}(u) + r_u \cdot \begin{cases} \frac{\mathrm{abs}(v)-\mathrm{abs}(u)}{|\mathrm{abs}(v)-\mathrm{abs}(u)|} & \text{if } n \text{ is a root node} \\ \frac{\mathrm{abs}(n_v)-\mathrm{abs}(u)}{|\mathrm{abs}(n_v)-\mathrm{abs}(u)|} & \text{otherwise} \end{cases}$$

$$\mathrm{abs}(v) = \begin{cases} 0 & \text{if } v = \mathit{root} \\ \mathrm{pos}(v) + \mathrm{pos}(\mathrm{par}(v)) & \text{otherwise} \end{cases}$$



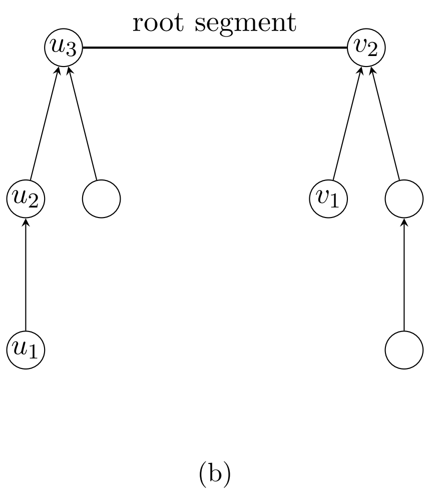


Figure 3.14: Projection of edges on the hierarchy and their corresponding segment trees.

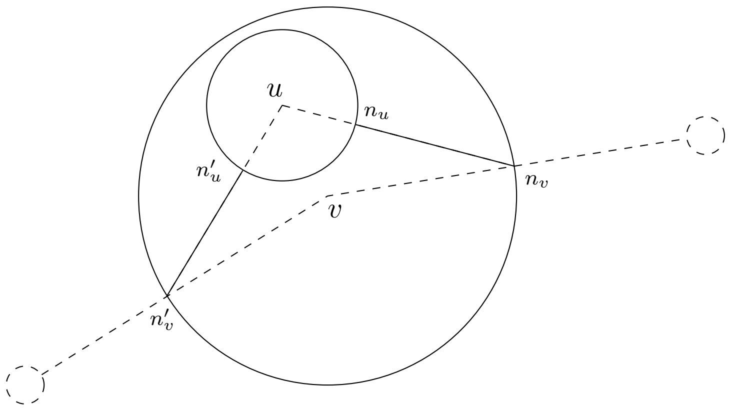


Figure 3.15: Positions for non-root nodes.

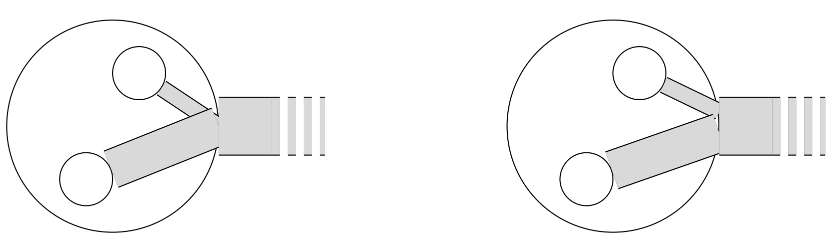

(a) Simple transition. (b) Corrected transition.

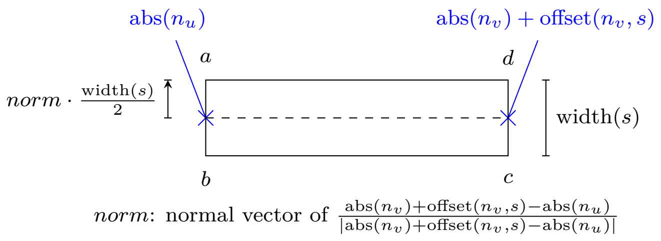
Figure 3.16: Transition between single segments and a thicker bundle.

With this definition for the positions of segment nodes, we can route each segment by inserting nodes at specific positions; these nodes are denoted as *intermediate* or *split nodes*. For finding appropriate positions, we first detect collisions between a segment and vertices. Since the partial layouts do not overlap, it is sufficient to compare the vertices of the partial graph enclosing the segment. Collisions can be determined by replacing segments with rectangles and computing the intersection between this rectangle and the disk representing the vertex; there exist plenty of suitable algorithms [37]. We should note that we compute only collisions with vertices not incident to a segment. Moreover, we introduce a metric for collisions to identify the vertex most obstructing for the respective segment. Finally, we insert a new node outside of the critical vertex and split the segment, connecting its end nodes by two new segments with the inserted node. Because the new segments might overlap with vertices as well, we repeat this process for the created segments until no collision is found. The complete procedure is outlined in Algorithm 3.6; with G_s and SG_s we refer to the partial graph and segment graph, respectively.

Algorithm 3.6 Segment routing

```
1: procedure ROUTE($s = (n_{\text{start}}, n_{\text{end}})$, $SG_s = (N_s, S_s, nv, se)$, $G_s = (V_s, E_s)$)
2:     $n_{\text{split}} \leftarrow$ FINDSEVERESTCRITICALITY($s$, $G_s$)
3:     if null = $n_{\text{split}}$ then
4:         return
5:     Remove $s$ from $SG_s$
6:     Add split node $n_{\text{split}}$ to $SG$
7:     Add $s_1 = (n_{\text{start}}, n_{\text{split}})$ to $SG$
8:     Add $s_2 = (n_{\text{split}}, n_{\text{end}})$ to $SG$
9:     ROUTE($s_1$, $SG_s$, $G_s$)                    ▷ $SG_{s_1} = SG_{s_2} = SG_s$
10:    ROUTE($s_2$, $SG_s$, $G_s$)
```

For computing the criticality we still have a slight problem since multiple segments leaving a vertex with the same target are bundled into one thicker segment. As seen in Figure 3.16a, using the same node position for all these segments spoils the transition between the non-bundled and the bundled segments. However, we can correct the transition by offsetting the incoming bundles as shown in Figure 3.16b.



$\text{abs}(n_u)$
$\text{abs}(n_v) + \text{offset}(n_v, s)$
a
d
$\mathit{norm} \cdot \frac{\text{width}(s)}{2}$
$\text{width}(s)$
b
c
norm: normal vector of $\frac{\text{abs}(n_v)+\text{offset}(n_v,s)-\text{abs}(n_u)}{|\text{abs}(n_v)+\text{offset}(n_v,s)-\text{abs}(n_u)|}$

Figure 3.17: Rectangle for a segment $s = (n_u, n_v)$.

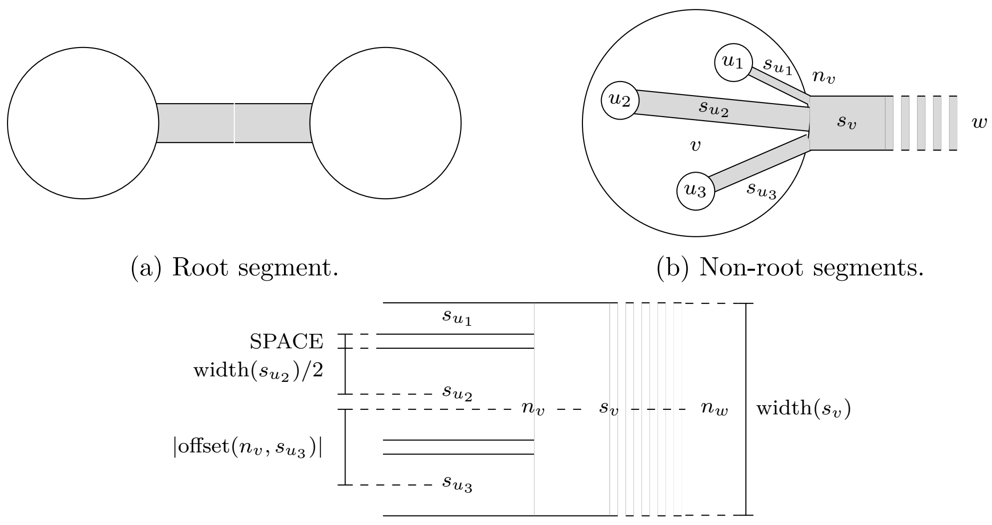
The transition is still not perfect, but we show how to create smooth transitions when we route single edges. This leads to slightly different edge routes, for which we do not prove that they are free of collisions. Nevertheless, in practice we did not experience collisions in layouts generated by our algorithm. Let $s = (n_u, n_v)$ denote a segment that is either a root segment or, without loss of generality, for which v is the unique parent of u. Furthermore, let *norm* denote the normalized normal vector for the vector from n_v to n_v as illustrated in Figure 3.17. Then we can compute the rectangle $\text{rect}(s)$ for s as follows:

$$\begin{aligned}
&\text{rect}(s) = (a, b, c, d) \\
&\qquad a = \text{abs}(n_u) + \mathit{norm} \cdot \frac{\text{width}(s)}{2} \\
&\qquad b = \text{abs}(n_u) - \mathit{norm} \cdot \frac{\text{width}(s)}{2} \\
&\qquad c = \text{abs}(n_v) + \text{offset}(n_v, s) - \mathit{norm} \cdot \frac{\text{width}(s)}{2} \\
&\qquad d = \text{abs}(n_v) + \text{offset}(n_v, s) + \mathit{norm} \cdot \frac{\text{width}(s)}{2}
\end{aligned}$$

The width for a segment s can be computed by the number of edges $|s|$ associated with it; the constants WIDTH and SPACE in the equation below denote the width of a single edge and the required space between two edges so that the edges can be drawn in parallel.

$$\text{width}(s) = |s| \cdot \text{WIDTH} + (|s| - 1) \cdot \text{SPACE}$$

Regarding the offset, we again have to differentiate between root and non-root segments. If s is a root segment, then the offset is zero as indicated in Figure 3.18a. If s is not a root segment, we can compute the offset using the idea given in Figure 3.18b and 3.18c. Let $s_{u_i} = (n_{u_i}, n_v)$ for $i \in \{1, ..., n\}$ denote the incoming non-root segments for a node n_v and $s_v = (n_v, n_w)$ the bundled segment leaving for another node n_w. Assuming



(a) Root segment.

(b) Non-root segments.

(c) Offset for non-root segments.

Figure 3.18: Determining the segment offset.

that the nodes n_{u_i} are sorted according to their angle towards n_v, the offset is the result of the following equations:

$$\text{offset}(n_v, s_{u_k}) = norm_{s_v} \cdot \left(\sum_{i<k} (\text{width}(s_{u_i}) + SPACE) + \frac{\text{width}(s_{u_k}) - \text{width}(s_v)}{2} \right)$$

$$norm_{s_v}\text{: normal vector of } \frac{\text{abs}(n_w) - \text{abs}(n_v)}{|\text{abs}(n_w) - \text{abs}(n_v)|}$$

The sorting of incoming nodes n_{u_i} is done as illustrated in Figure 3.19. For the rare case of nodes with the same angle – this can happen due to vertices being placed behind each other – we can simply use the distance of the nodes to the bundling node v_n as a secondary sorting criterion. Since vertices are not overlapping, this induces a unique order on the incoming nodes.

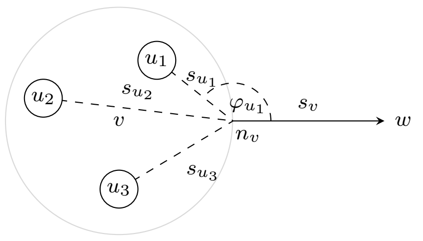


Figure 3.19: Sorting nodes radially.

Finally, we are able to determine the criticalities for segments. In Figure 3.20, a segment s represented by a rectangle and intersecting a few vertices is displayed. It is easy to see that even though the vertex u is bigger than the vertex v, the latter one is more obstructive. An intelligent way to route the segment around a colliding vertex v is to insert a split node n_{split_v} outside of v that lies on the line given by v and the reference point p_v. Assuming that we already know how to compute the position of the respective split nodes,

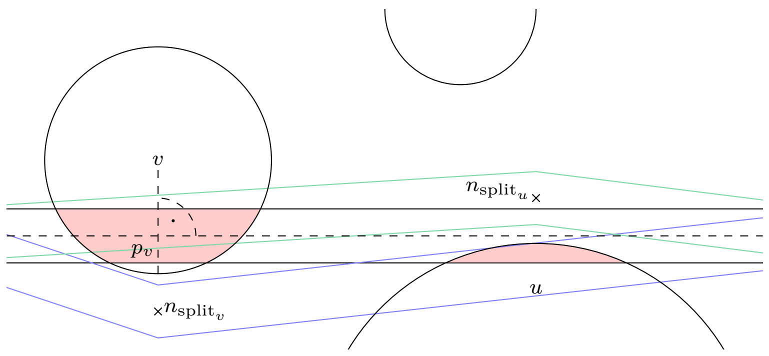


Figure 3.20: Critical points for a segment.

a simple metric to measure the severity of a criticality c_v for a vertex v is to compute the distance of the split node n_{split_v} to the collision's reference point p_v, that is, the deviation from the previous route:

$$sev_v = |\text{abs}(n_{\text{split}_v}) - p_v|$$

The severest criticality of a segment may then be computed as in Algorithm 3.7. Regarding the position of n_{split_v}, we use the equation given below; in the rare case that $p_v = \text{abs}(v)$, that is, the center of the vertex and the reference point coincide, one can use the normal unit vector of s. The formula ensures the distance between the split node and the vertex is great enough so that the new segments do not collide with the vertex at their end points. However, as illustrated in 3.21 there could still be collisions between the same vertex and the new segments that need to be removed.

$$n_{\text{split}_v} = \text{abs}(v) + \frac{p_v - \text{abs}(v)}{|p_v - \text{abs}(v)|} \cdot (r_v + \text{width}(s)/2 + SPACE + \min(r_v, \text{width}(s)))$$

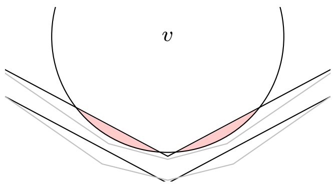


Figure 3.21: New collisions after routing a segment.

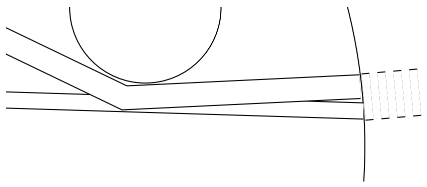


Figure 3.22: Overlapping neighboring segments.

3.5.5 Crossing Reduction and Rebundling

The problem with the given heuristic is that as illustrated in Figure 3.22 it may cause unnecessary crossings between neighboring segments with the same target. With target,

Algorithm 3.7 Finding criticalities

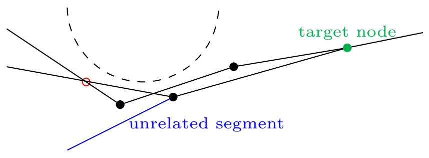
```
function FindSeverestCriticality($s$, $G_s$)
    $sev \leftarrow -\infty$
    $n_{\text{split}} \leftarrow$ null
    for each $v \in V_s$ do
        if intersect$(s, v)$ then
            $n_{\text{split}_v} \leftarrow$FindSplitNode($s$, $v$)
            $sev_v \leftarrow |\text{abs}(n_{\text{split}_v}) - p_v|$
            if $sev < sev_v$ then
                $sev \leftarrow sev_v$
                $n_{\text{split}} \leftarrow n_{\text{split}_v}$
    return $n_{\text{split}}$

function FindSplitNode($s$, $v$)
    $p_v \leftarrow ...$                                        ▷ computation of $p_v$ omitted
    $dir \leftarrow \frac{p_v - \text{abs}(v)}{|p_v - \text{abs}(v)|}$                     ▷ direction towards $p_v$
    if $|dir| = 0$ then
        $dir \leftarrow$ normal unit vector of $s$
    return $\text{abs}(v) + dir \cdot (r_v + \text{width}(s)/2 + SPACE + \min(r_v, \text{width}(s)))$
```

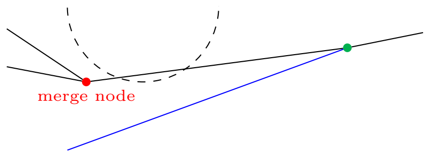
we do not mean the end node of the segments, but the first node at which the paths indicated by the segments join. The main reason for these crossings is that split nodes for segments with greater width are placed in greater distance from obstructing vertices. We eliminate these crossings by merging the involved segments roughly at the point at which they cross. The input for this step is a valid routing, that is, there are no vertices overlapping with segments.

The basic procedure is to create the *merge node*, connect it with a new segment to the first node that is passed by both segments, the *target node*, and remove obsolete segments and intermediate nodes. However, as seen in Figure 3.23 there may be another segment joining one of the merged segments before the target node; we have to connect these segments to the target node as well since the intermediate node will be removed. We remove the segments in between since the thickness of the merged segments increases and this might lead to new vertex collisions; afterwards, we again compute a routing for the new segments. In contrast, segments that are not subject to change do not have to be rerouted. As in the vertex layout and the segment routing, we can remove the crossings separately for each partial layout.

To check if two segments cross each other we simply compute the intersection of the rectangles introduced in the previous section. Nevertheless, we bundle only segments converging at some point, hence it is not necessary to check all segment pairs in the partial segment graph. We limit ourselves to the segments in the tree given by segments targeting the same exit node in a partial layout as exemplified in Figure 3.24. We do



(a) Before removal.



(b) After removal.

Figure 3.23: Unrelated segments joining segments that will be removed.

not go into detail about how to find crossings between segments efficiently, one could for example employ a modified Bentley-Ottmann algorithm [9].

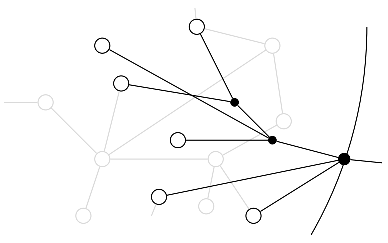
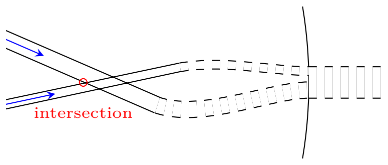
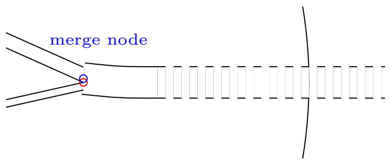


Figure 3.24: Segment tree for an exit node.

Now, we only need to explain how to place the merge node to remove a crossing. A possible solution is to replace the segments with simple lines and use the nearest point between them as merge point. These lines do not necessarily cross, so it is not sufficient to just compute their intersection. However, if these lines cross, we encounter another issue: On the intersection point, the segments overlap completely, and after a merge at this point they would use less space than before. To avoid this, we choose the nearest point between the inner lines of a segment. For the rectangles of two segments, the inner lines are the sides that overlap first if we move towards the final merge node on the border of the partial layout; an example is given in Figure 3.25a. Furthermore, we correct the merge position respective to the width of the involved segments as demonstrated in Figure 3.25b; we omit the geometric details at this point. Finally, in Algorithm 3.8 the complete procedure of finding and removing an intersection between segments is sketched.



(a) Inner lines of the rectangles representing the segments.



(b) Merge node with offset.

Figure 3.25: Merge node computation.

3.5.6 Complete Segment Routing

The two steps described before lead us to Algorithm 3.9. We compute the layout separately for each partial graph and employ a queue to save segments for which we have to find a routing and remove crossings. We initialize the queue by adding all segments of the partial layout and process them in two steps. First, we route and remove all seg-

Algorithm 3.8 Crossing removal

```
procedure RemoveCrossing($SG_w = (N_w, S_w, nv, se)$, $G_w = (V_w, E_w)$)
    $s_1 = (n_{u_1}, n_{v_1}), s_2 = (n_{u_2}, n_{v_2}) \leftarrow ...$        ▷ find a crossing in $SG_w$, $G_w$
    if no crossing found then
        return
    $n_{\text{merge}} \leftarrow ...$        ▷ merge node for $s_1$ and $s_2$
    $n_{\text{target}} \leftarrow ...$        ▷ target node for $s_1$ and $s_2$
    $S_{\text{obs}} \leftarrow ...$        ▷ segments leading from $u_1$ and $u_2$ to $n_{\text{target}}$
    $N_{\text{obs}} \leftarrow \{n | n \text{ is an end node for } s \in S_{\text{obs}}\} \setminus \{n_{u_1}, n_{u_2}, n_{\text{target}}\}$
    $S_{\text{inc}} \leftarrow \{(n_u, n_v) \in S_v \setminus S_{\text{obs}} | n_v \in N_{\text{obs}}\}$
    remove $S_{\text{obs}} \cup S_{\text{inc}}$ from $S_w$
    remove $N_{\text{obs}}$ from $N_w$
    add $n_{\text{target}}$ to $N_w$
    add $(n_{u_1}, n_{\text{merge}})$, $(n_{u_2}, n_{\text{merge}})$ and $(n_{\text{merge}}, n_{\text{target}})$ to $S_w$
    for each $(n_u, n_v) \in S_{\text{inc}}$ do
        add $(n_u, n_{\text{target}})$ to $S_w$
```

ments in the queue; in this process, new segments may be inserted due to old segments being split up. When the queue is empty, crossings in the partial layout are removed, if possible. Since the route of the segments might be modified by merging segments in this second step, we have to repeat the complete process as long as segments remain to be routed or an unnecessary crossing is found. Even though we omitted the queue in the previous algorithms, we assume that any new segment is added to the queue by the respective subprocedure in which it is created. In the end, we obtain a layout in which no segment overlaps with a vertex.

Algorithm 3.9 Segment routing

```
procedure Route($SG = (N, S, nv, se)$, $G = (V, E, root, H)$)
    for each $v \in V$ do        ▷ segment routing in every partial layout
        $G_v = (V_v, E_v) \leftarrow$ partial graph for $v$
        $SG_v = (N_v, S_v, nv, se) \leftarrow$ partial segment graph for $v$
        $Q \leftarrow S_v$        ▷ initialize segment queue
        while $Q$ is not empty do        ▷ compute segment routing
            while $Q$ is not empty do
                $s \leftarrow$ pop($Q$)
                Route($s$, $SG_s = SG_v$, $G_s = G_v$, $Q$)
            RemoveCrossing($SG_v$, $G_v$, $Q$)
```

3.5.7 Edge Routing

After routing the segments and removing all unnecessary crossings, we can draw single edges. Most of their route is already fixed by the segment routes, we only have to draw

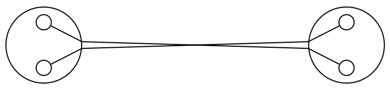


Figure 3.26: Inevitable crossings.

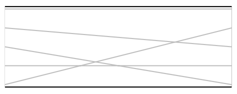


Figure 3.27: Edges within a segment.

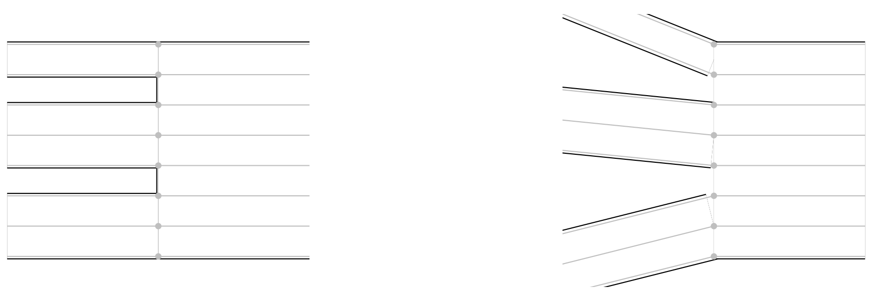
them properly within their segments. Therefore, we have three requirements: Edges should be mainly confined to the space assigned to their segments, they have to be connected at segment transitions and we want to minimize the number of edge crossings within segments. As seen in Figure 3.26 we cannot completely avoid crossings; however, we show that it is possible to draw non-root segments without crossings and we reduce the number of crossings within root segments.

If we start with a simple segment, we can draw the edges by choosing a start and an end position for each of them and connecting these points with a straight line as seen in Figure 3.27. The position can be computed according to the width of the respective segment; before, we already reserved space such that the edges could be drawn in parallel without touching. Let i denote the index of an edge e in a segment $s = (n_u, n_v)$, then its start position can be computed using the start node as base as in the formula below. Here, dir denotes the normal unit vector for the vector from n_u to n_v. The end positions can be obtained by permuting the indices and applying the same equation with the nodes of the segment switched.

$$\text{start}(s = (n_u, n_v), i) = |\text{abs}(n_u)| + dir \cdot (i \cdot \text{WIDTH} + (i - 1) \cdot \text{SPACE} - \frac{1}{2}\text{width(s)})$$

For connecting edges at segment transitions, we just have to ensure that the end positions of one segment are consistent with the start position of the other. Recalling the segment tree defined before, it is easy to see that the start positions for leaf segments may be chosen freely since they do not depend on incoming segments. Assuming that we fixed the start positions, we can choose the end positions accordingly such that all edges are drawn in parallel and hence without crossings. Next, we can determine the positions for bundling segments by deriving them according to their incoming segments to connect the edges as illustrated in Figure 3.28; again, the edges are parallel to each other. Of course, segments usually arrive in different angles, but if we use the start positions of the bundle segment for the end positions of the incoming segments, we achieve a drawing with suitable transitions that is still free of crossings at the cost of slightly distorted rectangles. We continue this up to the root segment, for which we have to derive the positions of incoming edges at both sides. The root segment is also the only segment at which crossings between internal edges may occur.

The last step is to reduce the amount of crossings within the root segment by changing the order of the edges at the leaf segments. This also induces a different edge order in all other segments. We found two types of edge crossings that can easily be avoided: Intersections of multiple edges between the same vertices and crossings of edges originating from the same vertex. Most, if not all other crossings are caused by the routing of the



(a) (b)

Figure 3.28: Segment transitions.

segments and thus cannot be prevented. The first case is illustrated in Figure 3.29 and, provided that the edges are sorted equally for both vertices, can be fixed by inverting the order at which edges leave a vertex for one of the two vertices. Supposed that a unique index is assigned to each vertex, we can decide whether to invert the order for a vertex or not by simply comparing it with its current counterpart.

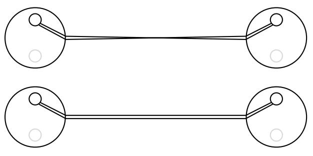


Figure 3.29: Crossings between multiple edges.

An example for the second case is given in Figure 3.30: As illustrated, it is possible to draw the edges of a segment s ending at the vertex v such that they never cross each other. To achieve this, it is not only necessary to invert the order for multiple edges, but also to sort the edges according to the segment routing. A suitable order can be computed as described in Algorithm 3.10. We start with the root segment s_r and determine the partial order for each incoming segment on the opposing side of v recursively. Segments without incoming segments are end segments and we can draw all involved edges in parallel as denoted before. For other segments, we form the complete order by assembling the partial orders of their incoming segments in radial order. We already used this order to find an appropriate offset for connecting segments, an example was given in Figure 3.19 on page 50. Finally, we are able to compute all the remaining edge positions – and thus the complete layout – as detailed in the first part of this section, the start positions at the leaf segments are derived from the order given by this algorithm.

3.6 Conclusion

In this chapter, we described how to compute a complete layout for a hierarchic graph using force-directed methods and routing heuristics. We omitted proofs and cannot guarantee that the algorithm produces a collision-free layout corresponding to our requirements. It is also possible that the heuristic given for routing the edges does not terminate. However, even though our algorithm is not very efficient so far, there is room for improvement and it is effective; as explicated in a later chapter, the layouts we created are of high quality.

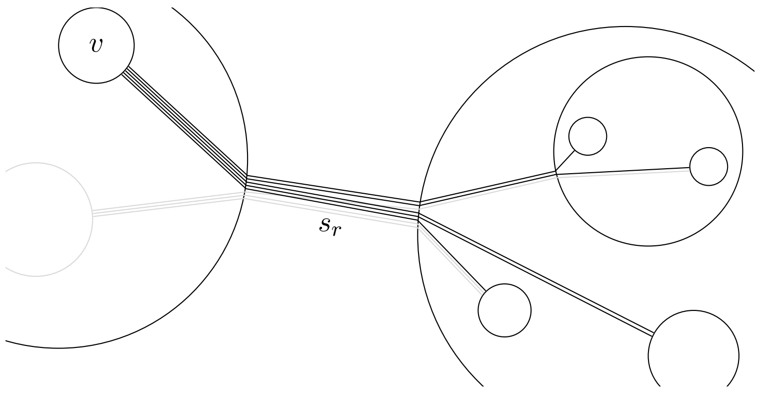


Figure 3.30: Edges originating at the same vertex.

Algorithm 3.10 Edge order

```
1:  function EdgeOrder($v$, $s$)
2:      $e_v \leftarrow$ ...                                  ▷ edges ending at $v$
3:      $S_{\text{inc}} \leftarrow$ ...                       ▷ incoming segments opposite to $v$
4:      if $S_{\text{inc}} = \varnothing$ then
5:          $u \leftarrow$ ...                                ▷ end vertex of $s$
6:          $e_{v \cap u} = e_v \cap e_u$                     ▷ edges between $u$ and $u$
7:          if $\text{index}_u < \text{index}_v$ then
8:              return list of $e_{v \cap u}$ in arbitrary order
9:          else
10:             return list of $e_{v \cap u}$ in inverted order
11:     $\text{list}_e \leftarrow \varnothing$
12:     for each $s_{\text{inc}} \in S_{\text{inc}}$ do       ▷ assume $S_{\text{inc}}$ sorted according to their angles
13:         $\text{list}_{e_s} \leftarrow$ EdgeOrder($v$, $s_{\text{inc}}$)
14:         $\text{list}_e \leftarrow \text{list}_e + \text{list}_{e_s}$
        return $\text{list}_e$
```

4 Parameter Studies

In this chapter we detail the effect of the different parameters for our algorithm and derive values that lead to good layouts. We start with basic factors that primarily determine local correlations of vertices and edges, that is, attraction and repulsion. Next, we deal with the gravity needed for disconnected graphs and discuss how to choose the temperature and the number of iterations. Finally, we briefly recapitulate our findings and enumerate all parameters with suitable values.

Before we go any further, there are three parameters that dictate the minimum scale of our layout, namely edge width, edge spacing and the minimum radius for vertices. Clearly, if the space between edges or the minimum radius is much greater than the edge width, the layout is destroyed since it is hardly possible to see edges between vertices. Having said that, if we regard this, their impact on the layout quality is insignificant so we do not address them further. Our layouts are computed with 1 for the edge width, 4 for the edge spacing and 20 for the minimum radius.

4.1 Attraction and Repulsion

The most important parameters are the ones concerning repulsive and attractive forces because they determine the local layout and thus also the global layout; the respective equations are displayed below. We have to consider three major factors: The stability of the forces, the general distance between vertices and the weight of the radii.

$$\begin{aligned}
d_X(u,v) &= r_u + r_v + b_X \cdot (c_{\min}\min(r_u,r_v) + c_{\max}\max(r_u,r_v)) \\
&\quad X \in \{\min, \text{pref}\} \\
&\quad b_{\text{pref}} > b_{\min} \geq 1 \\
&\quad c_{\min}, c_{\max} \geq 0 \\
&\quad c_{\min} + c_{\max} = 1 \\
d'_X(r_u,r_v) &= d_X(r_u,r_v) - d_{\min}(r_u,r_v), X \in \{\text{pref}, \text{actual}\} \\
f_{\text{rep}}(r_u,r_v) &= \begin{cases} \frac{d'_{\text{pref}}(r_u,r_v)^2}{d'_{\text{actual}}(r_u,r_v)} & \text{if } 0 \leq d'_{\text{actual}}(r_u,r_v) \\ \infty & \text{otherwise} \end{cases} \\
f_{\text{attr}}(r_u,r_v) &= \begin{cases} \frac{d'_{\text{actual}}(r_u,r_v)^2}{d'_{\text{pref}}(r_u,r_v)} & \text{if } 0 \leq d'_{\text{actual}}(r_u,r_v) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

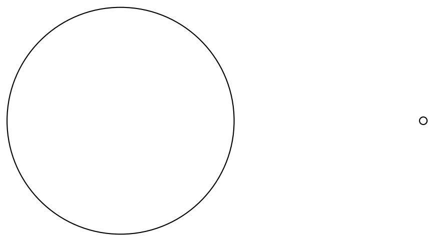
By the term stability of forces we refer to the behavior of vertices that are moving towards each other. If their distance falls below the minimum distance, the attractive force becomes zero and the repulsive force infinity resulting in a strong force pulling them apart. However, the closer b_{pref} is to b_{min}, the weaker the repulsive force becomes and the stronger the attractive force because $d'_{\text{pref}}(r_u, r_v)$ approaches zero. This means that whenever the actual distance is bigger than the minimum distance, the vertices are strongly contracted, and the vertices are likely to oscillate without finding a stable position during the algorithm. The solution for this is to increase the size of the window, that is, the size of b_{min} in comparison to b_{pref}.

$$d'_{\text{pref}}(r_u, r_v) = (b_{\text{pref}} - b_{\text{min}}) \cdot (c_{\text{min}} \min(r_u, r_v) + c_{\text{max}} \max(r_u, r_v))$$

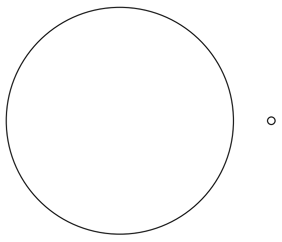
The next issue is the distance between vertices that depends on the size of b_{pref} and b_{min}. If these parameters are too large, we obtain a layout where the vertices are strongly scattered and it becomes difficult to spot vertices in deeper levels. In contrast, the layout clogs in case we choose a small value for these parameters so that there is not enough space between the vertices to route the edges; colliding edges are highlighted in red. As illustrated in Figure 4.2, $b_{\text{min}} \in [2, 6]$ and $b_{\text{pref}} = 2b_{\text{min}}$ lead to the best results in our experiments.

The last parameter pair, c_{min} and c_{max}, is only important if vertices of strongly divergent size are member of the same partial graph. If we weigh the radii equally or favor the bigger one, a layout as in Figure 4.1a forms. However, the layout of Figure 4.1b, where smaller vertices are placed nearer to bigger vertices, is much more space-saving. This is especially important for graphs with deep hierarchies since the size of child vertices is propagated to their parents. We accomplished to generate a balanced layout by setting $c_{\text{min}} = 0.05$ and $c_{\text{max}} = 0.95$.

These four parameters are essential for the layout quality. For instance, if there is not enough space between vertices to route the edges, it is possible to adjust the parameters to distribute them on a larger area.

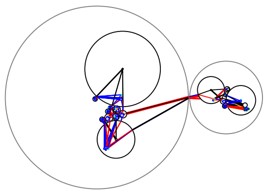


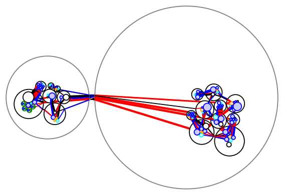
(a) $c_{\text{min}} \leq c_{\text{max}}$

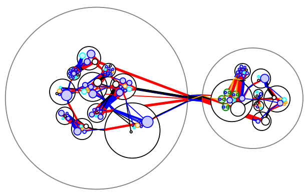


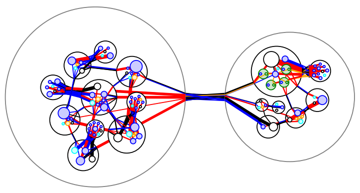
(b) $c_{\text{min}} >> c_{\text{max}}$

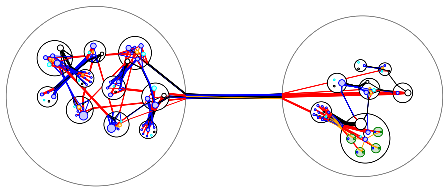
Figure 4.1: Parameter configurations for the ratio between vertex radii.

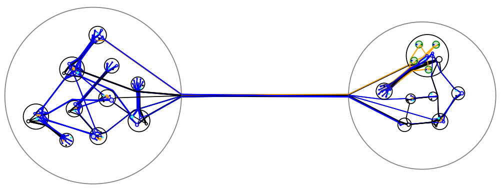

(a) $b_{\mathrm{min}} = 0.0625$

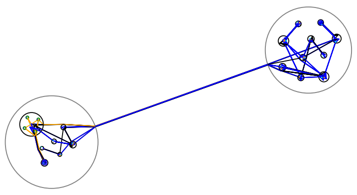

(b) $b_{\mathrm{min}} = 0.125$

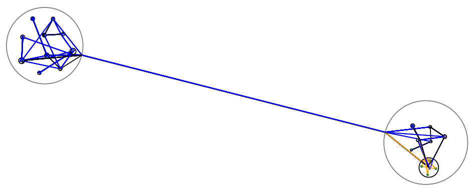

(c) $b_{\mathrm{min}} = 0.25$

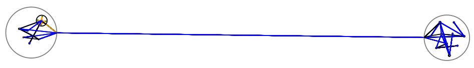

(d) $b_{\mathrm{min}} = 0.5$


(e) $b_{\mathrm{min}} = 1$


(f) $b_{\mathrm{min}} = 2$


(g) $b_{\mathrm{min}} = 4$


(h) $b_{\mathrm{min}} = 8$


(i) $b_{\mathrm{min}} = 16$

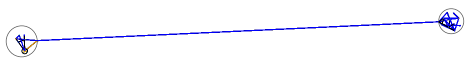

(j) $b_{\mathrm{min}} = 32$

Figure 4.2: Parameter configurations for the vertex distance with $b_{\mathrm{pref}} = 2b_{\mathrm{min}}$.

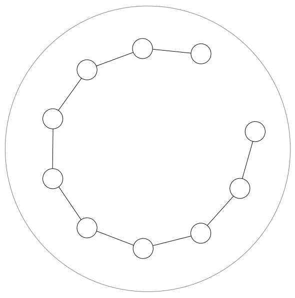


Figure 4.3: Compaction by gravity.

4.2 Gravity

The third force, the gravity, ensures that our layout does not collapse, if the graph is disconnected, and that it is compacted. It is crucial that the force is neither too weak nor too strong, otherwise the layout disintegrates or implodes as illustrated in Figure 4.4a and 4.4c. The parameter p determines the scalability of the gravity: The higher it is, the faster the gravity will grow with the layout size. The parameter g is merely a tuning factor that ensures that the gravity is adequate for the layouts on the lowest levels of the hierarchy. We obtained the best results with a quadratic gravity; the layouts displayed in Figure 4.3 and 4.4b were generated using $g = 0.001$ and $p = 2$.

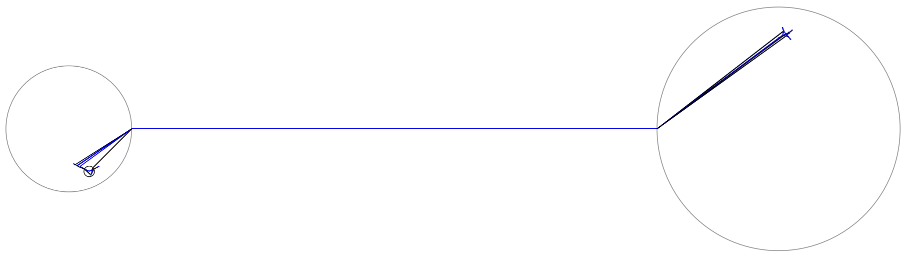
$$f_{\text{grav}}(v) = g \cdot |\text{pos}(v)|^p, g > 0, p \geq 0$$

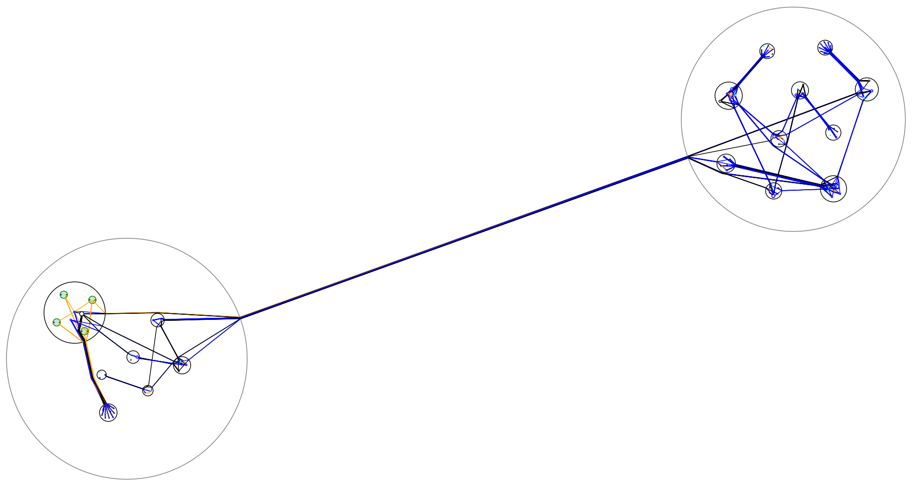
4.3 Number of Iterations and Temperature

The last parameters concerning the vertex layout are the temperature and the number of iterations. For a vertex v, the step size is restricted by $t \cdot r_{\text{par}(v)}$, that is, the temperature times the radius of its parent. Since the parent vertex encloses all of its children, each child can be moved to any other position within its parent if $t \geq 2$, hence it is sufficient to choose $t < 2$ as initial temperature. In fact, the layout seemed to deteriorate for a temperature near 2; we found that starting with $t = 1$ leads to stable results. Furthermore, we selected a function decreasing linearly over the number of iterations leading us to the function given below with $t_{\text{start}} = 1$.

$$t(i) = t_{\text{start}} \cdot \left(1 - \frac{i}{n}\right), i = 1, ..., n$$

Finally, we have to fix the number of iterations denoted by n. While a few hundred steps did not always lead to good results, after a few thousand steps the layouts stabilized; in Figure 4.5 the effect is illustrated. It is important to remark that the figures do not show one execution of the algorithm at different steps: Each figure represents a complete run of the algorithm with the respective value of n.


(a) $g = 0.1$, $p = 1$


(b) $g = 0.001$, $p = 2$

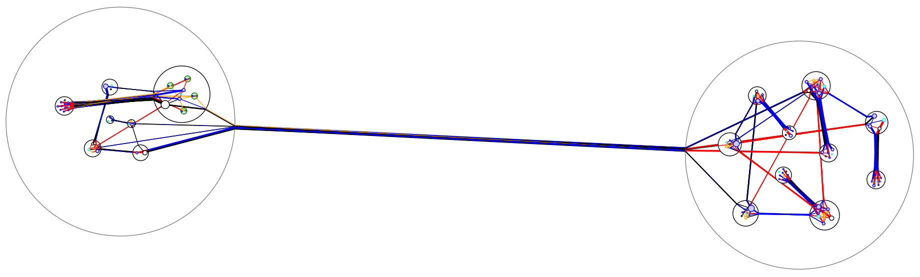
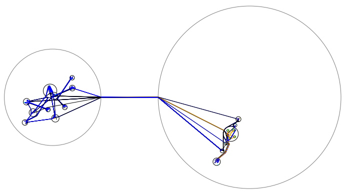
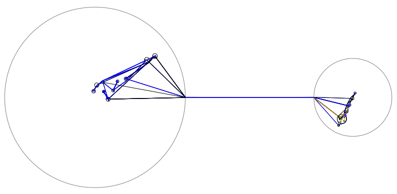
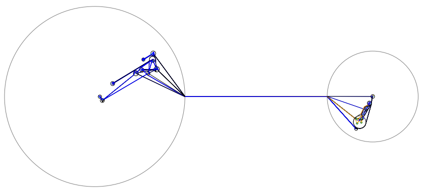
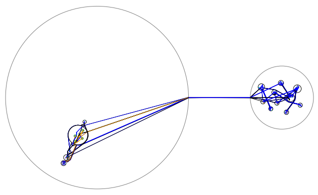

(c) $g = 1$, $p = 2$

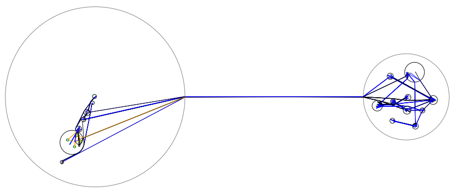
Figure 4.4: Gravity parameter configurations.

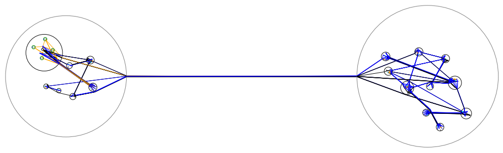

(a) $n = 10$

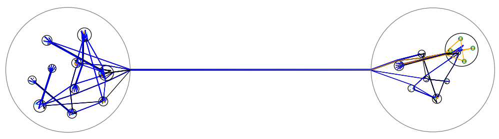

(b) $n = 20$

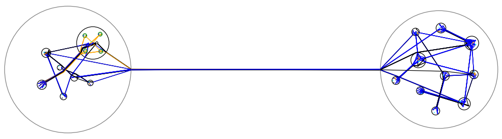

(c) $n = 40$

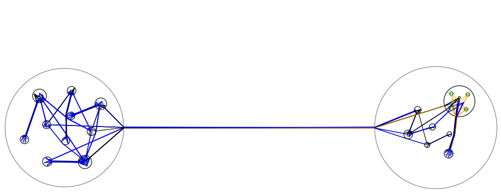

(d) $n = 80$


(e) $n = 160$


(f) $n = 320$


(g) $n = 640$


(h) $n = 1280$


(i) $n = 2560$

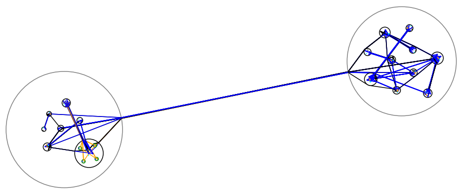

(j) $n = 5120$

Figure 4.5: Number of iterations.

4.4 Summary

We applied our algorithm to different real-world instances and succeeded in generating nice-looking layouts by employing the parameter set introduced before; all the parameters are listed in Table 4.1. Sometimes it was necessary to adjust the parameters for the vertex distance in order to enable the edge routing, but in most cases we could use the exact same set.

Parameter	**Letter**	**Value**
basics	WIDTH	1
	SPACE	4
	RADIUS	20
$f_{\mathbf{rep}}$, $f_{\mathbf{attr}}$	b_{min}	2-6
	b_{pref}	$2b_{\text{min}}$
	c_{min}	0.95
	c_{max}	0.05
$f_{\mathbf{grav}}$	g	0.001
	p	2
iterations	n	1000-5000
temperature	t_{start}	1

Table 4.1: Parameters for the layout algorithm.

5 Evaluation

Before concluding our work, we evaluate the results of our research in this chapter. First, we present our results, and afterwards, we show some shortcomings in our approach and discuss options for improving the algorithm in the future.

5.1 Results

We conducted experiments on one artificial project, the mentioned formula parser, and a few real-world instances. The algorithm designed in this work is capable of creating high-quality layouts for each of them. We start by reviewing the layouts of our example project in detail and continue by touching on the other projects. Finally, we introduce two prototypes for tools employing our algorithm that can be used to analyze software.

5.1.1 Example Projects

For the two hand-drawn example layouts we also created an automatic layout, which is pictured in Figure 5.2a and 5.2b, respectively. Even though the automatic layouts do not reflect the symmetries as nicely as the manual layouts, they still resemble each other and it is possible to find the code duplication in the automatic layout as well. Besides, they preserve a high level of detail: In Figure 5.1, a section of the refactored graph is magnified. Here, class vertices and their representative vertices are colored in black, method vertices in blue, field vertices in orange and constructor vertices in teal. The different edge types are also distinguished; black edges are references, class extension or interface implemention, blue edges represent method calls and orange edges indicate field accesses.

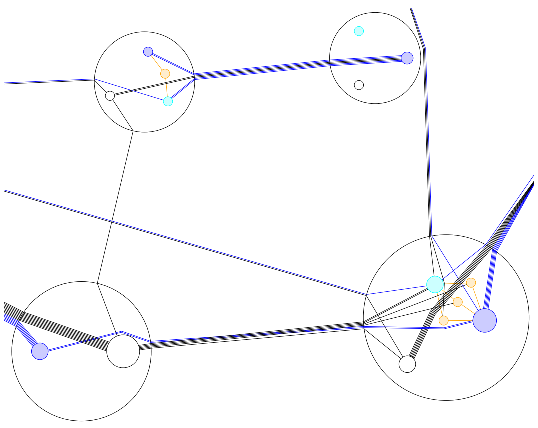
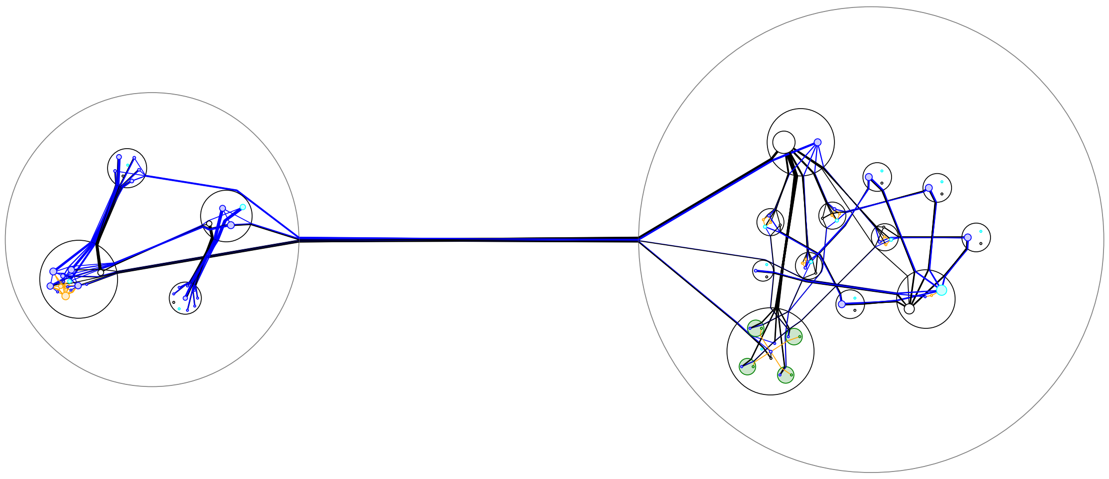


Figure 5.1: Detailed section of the refactored layout.

It is easy to understand that the layouts strongly reflect the structure originally imposed by the developer. For instance, the program logic of the formula parser is distributed on two packages: While the left one deals with the parsing of actual text, the right one models formulas mathematically. Furthermore, this finding is not only limited to our artificial example, but also applies to other real-world projects. In Figure 5.3, a


(a) Original project.

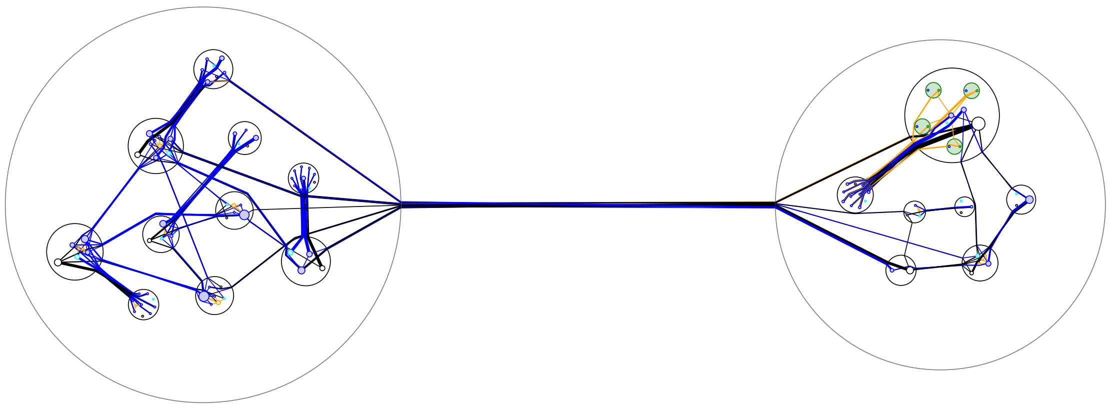

(b) Refactored project.

Figure 5.2: Automatic layout of the formula parser.

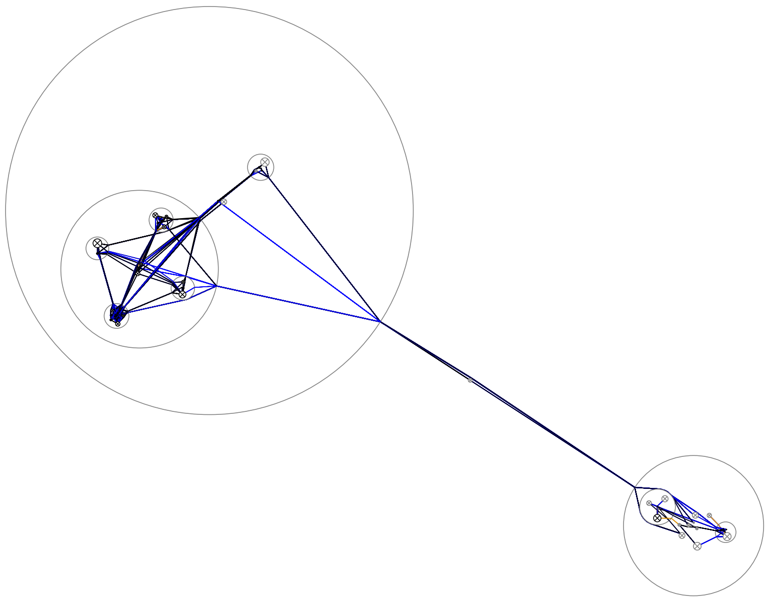


Figure 5.3: Project A.

small command-line tool is displayed that is split into two parts, namely the application and framework classes that are not specific to the code. An even better example is the project illustrated in Figure 5.4; its three main components are the graphical user interface, the domain model and utility classes. Finally, a software written according to the model-view-controller pattern is shown in Figure 5.5. The small external disk contains only framework classes and the biggest, which represents the application, consists of a model package, a package for the view and one for the controller.

5.1.2 Tooling

In addition to designing the algorithm, we also developed a tool so that we could employ the algorithm to real-world software. In general, it provides an interactive visualization of the graph that allows pan and zoom; Figure 5.6 is a screenshot of the application. Furthermore, as illustrated in Figure 5.7, the user is able to select a vertex to highlight it together with its associated edges. Because very thin lines vanish if the user zooms out, vertices and edges are drawn with thicker or thinner lines depending on the current scale so that it is possible to recognize connections between vertices even though they are quite small. Of course, on a large scale single edges may no longer be visible. As stated previously in this work, we replaced the straight lines by Bézier curves to smoothen the layout; however, they may induce extra collisions with vertices.

In the next step, we integrated our tool into Eclipse, a well-known IDE, as shown in Figure 5.8. By this means we can link the source code directly with its graphical representation, which helps us, for example, to quickly grasp the inherent structure of

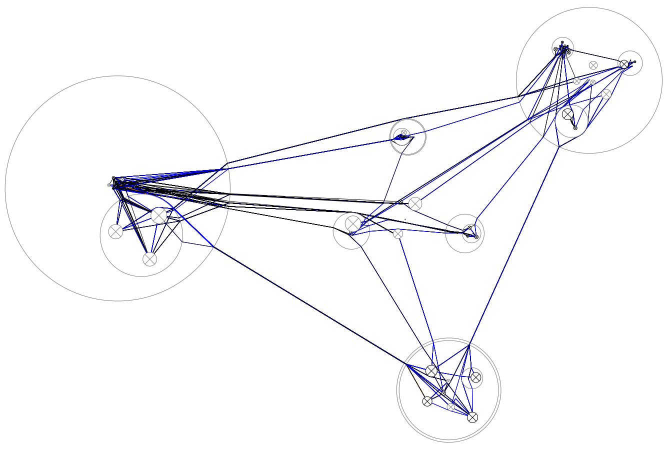


Figure 5.4: Project B.

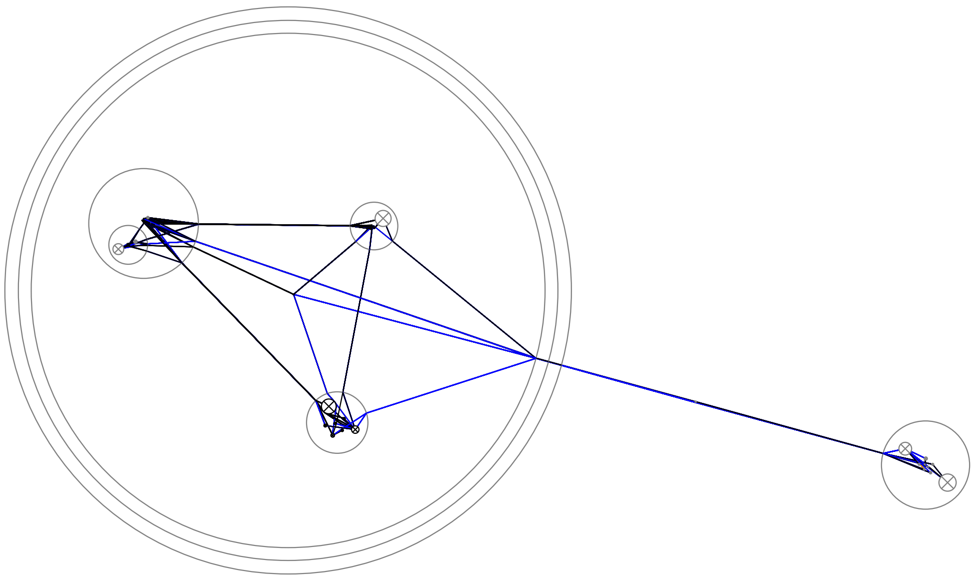
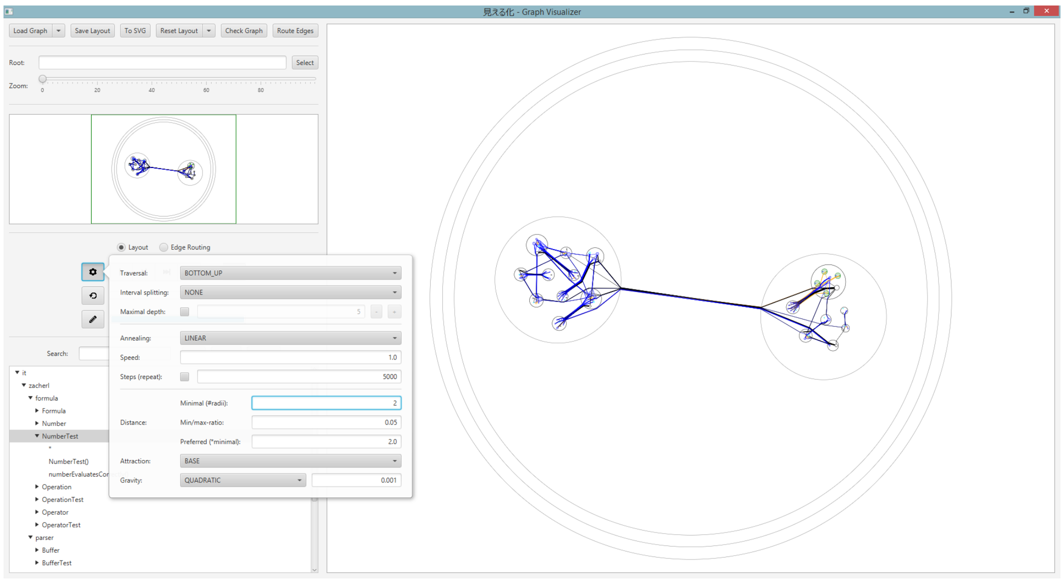


Figure 5.5: Project C.



Traversal: BOTTOM_UP
Interval splitting: NONE
Maximal depth: 5
Annealing: LINEAR
Speed: 1.0
Steps (repeat): 5000
Distance: Minimal (#radii): 2
Min/max-ratio: 0.05
Preferred (*minimal): 2.0
Attraction: BASE
Gravity: QUADRATIC 0.001

Figure 5.6: Tool for exploring a software graph.

a class or to inspect its external dependencies without having to read the complete file. Moreover, we can use it to display additional information like software metrics.

5.2 Future Work

Even though our algorithm provides satisfying results, there are still some deficiencies. Furthermore, we found different problems that have to be solved before we can develop functional tools. In this section, we discuss weak spots in the algorithm, address the challenges concerning the rendering of the layouts and enumerate opportunities for improving the tools.

5.2.1 Algorithm

While we focused on the effectiveness of the algorithm, we did not try to improve its performance as long as the layouts were generated in a reasonable amount of time. However, especially for very large graphs it would be beneficial to analyze and optimize the time complexity of the algorithm. Especially the heuristic for the edge routing should be improvable.

Apart from that, there are more specific opportunities to enhance the algorithm. Our initial layout is by far the least time-consuming part of the algorithm; nevertheless, a more subtle approach might reduce the number of iterations needed during the vertex layout without significantly prolonging the computation. One could even think about combining the initial layout and the vertex layout in one step by applying more sophisticated techniques like GRIP [24] instead of simple force-directed methods.

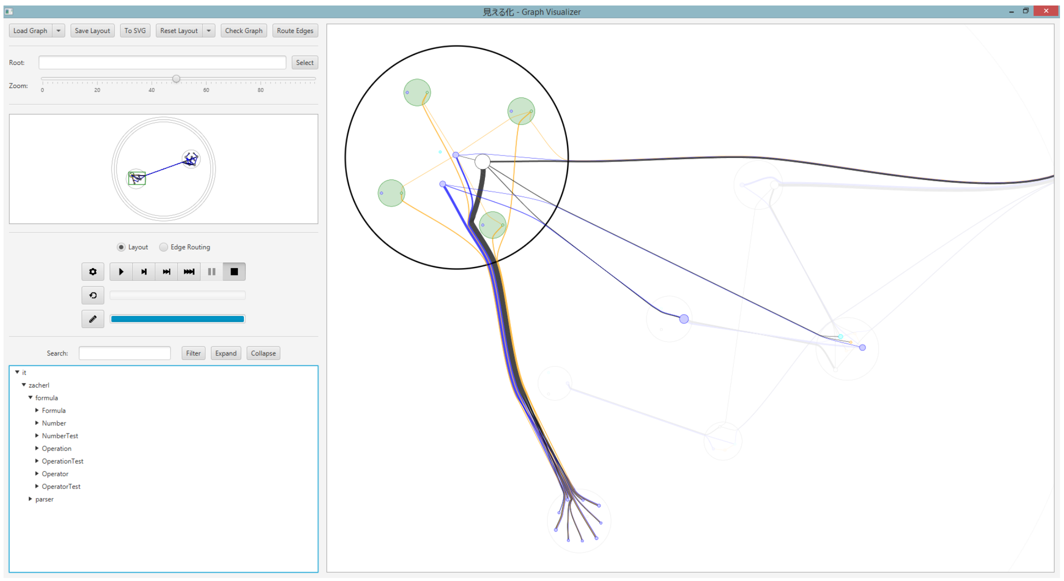
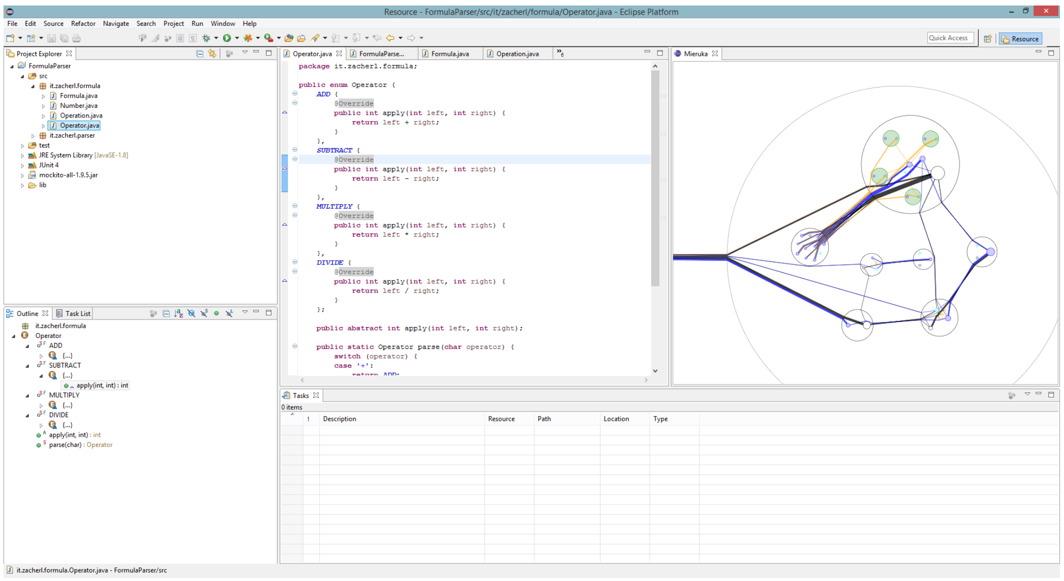


Figure 5.7: Highlighting of a vertex and its corresponding edges.



```
package it.zacherl.formula;

public enum Operator {
    ADD {
        @Override
        public int apply(int left, int right) {
            return left + right;
        }
    },
    SUBTRACT {
        @Override
        public int apply(int left, int right) {
            return left - right;
        }
    },
    MULTIPLY {
        @Override
        public int apply(int left, int right) {
            return left * right;
        }
    },
    DIVIDE {
        @Override
        public int apply(int left, int right) {
            return left / right;
        }
    };

    public abstract int apply(int left, int right);

    public static Operator parse(char operator) {
        switch (operator) {
        case '+':
```

Figure 5.8: Integration of the tool in Eclipse.

As for the quality of the layout, we ignore external edges while placing the vertices. A simple and not invasive way to improve the layout would be to rotate vertices so that the distance of their children's external edges is minimized. Admittedly, its effectiveness is restricted since the relative placement of the children towards their parent is not changed. A more promising method would be to apply the forces also to external edges or to introduce different forces for them, but this method is not only more complicated, but also causes different partial layouts to depend on each other.

Another mentionable issue concerns the gravity. As illustrated in Figure 5.9, we found that in some cases vertices consume a lot of space even though they are mainly empty; this results from the algorithm we currently employ to determine the radius of the parent vertex. Instead, we could also compute the minimal enclosing disk for all children and move them so that the center of the minimum enclosing disk coincides with the position of the parent vertex. This does not change the positions of the children relative to each other and thus the internal layout is not affected. However, if external edges are involved in the computation, the forces become hard to control and might oscillate. We hope that these issues can be eliminated for example by using an adaptive gravity.

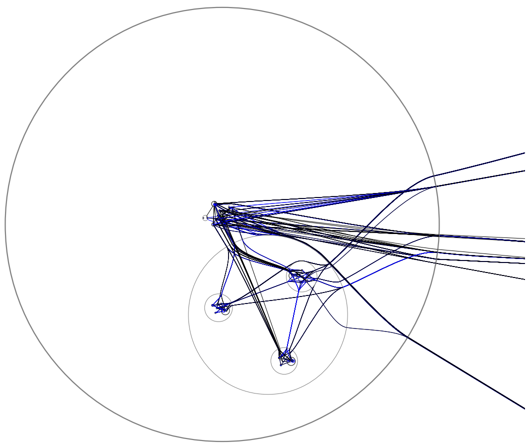


Figure 5.9: Vertex with suboptimal use of space.

The next topic we want to touch on is the edge routing. We believe that crossings between different edge bundles can be reduced further, whether or not these crossings occur at the border of vertices as shown in Figure 5.10 or not. In this context one could also try to bundle independent bundles that overlap or take similar routes. Another major issue is the space between vertices: We already mentioned that in some cases it is necessary to increase it so that all edges can be routed. Certainly, in the current form of the algorithm it is only possible to increase the space globally for all vertices, but most space problems in edge routing are limited to single vertices. Since the partial layouts can be computed independently, one could adjust the parameters of the force-directed algorithm locally so that the global layout remains compact.

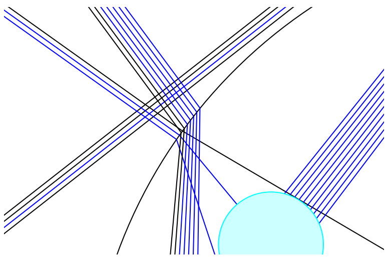


Figure 5.10: Edge bundles overlapping while leaving a vertex.

Finally, due to the modular structure of our algorithm we can easily replace parts as the edge routing by different techniques, and it might even be feasible to combine the vertex placement and the edge routing into one procedure. Furthermore, it would be interesting

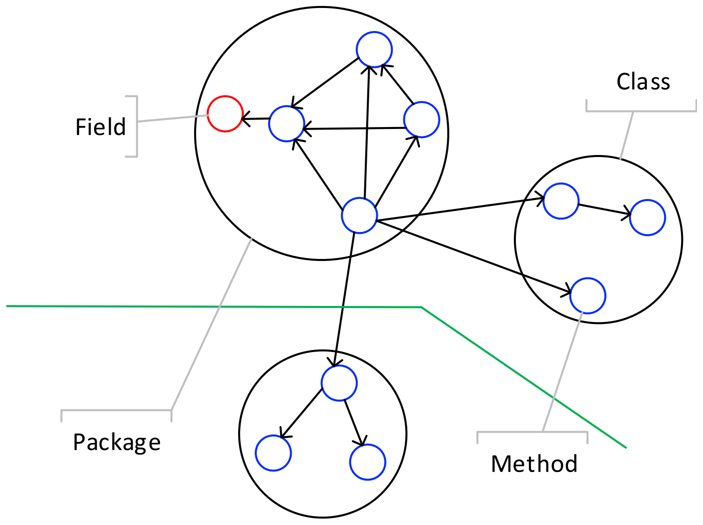


Figure 5.11: Class-centered layout for a hierarchic reference graph.

to combine various techniques for arranging the higher levels of the hierarchy differently. In Java, elements like classes usually possess a high cohesion, but the content of packages may vary strongly. For this reason, it might prove useful to ignore packages within the hierarchy so that types – excluding internal types such as nested classes – can be placed arbitrarily. As illustrated in Figure 5.11, packages could be visualized by partitioning the resulting layout using, for example, Voronoi diagrams, while classes, methods or fields are displayed hierarchically. In contrast to the pure hierarchic approach, this kind of visualization does not guarantee that packages are drawn monolithically.

Again, the map metaphor is reflected nicely. However, since child packages use the complete space of their parent packages, new difficulties emerge; an example would be how to indicate the hierarchy of packages.

5.2.2 Rendering

As from now, the biggest obstacle that prevents us from developing a nicely working tool is not the computation of the layout, but the visualization of it. If we provide an interactive map for exploring the software, we have to adjust the layout to the currently displayed section of the graph every time something changes; this also includes zooming and panning and something like a user selection. These changes may occur repeatedly within seconds or milliseconds and because the user interface has to respond immediately, it is too slow if one just tries to render the complete layout. These problems are common to geographic maps and it should be possible to apply their solutions in this context.

5.2.3 Tooling and Analysis

The last issue we want to bring up is the application of our findings. We already developed a prototype for exploring software and integrated it into an IDE, but it provides only basic functionality. For instance, it would be nice to display metrics on top of the layout as to analyze the overall condition of the software. Another interesting question is whether one can identify problems in the software by automatically analyzing the graph; the code duplication we found in our example project manifested itself in multiple identical structures and should be easily recognized.

6 Conclusion

In this thesis, we studied how to create a general-purpose visualization of software projects represented as graphs. While there already are a lot of different visualization techniques for software, these usually serve a specific purpose and do not intend to provide universal insights. We chose graphs for their capabilities of describing the structures in software and because it is simple to draw them two-dimensionally. For the visualization itself, we employed principles known from cartography since geographic maps are easily understandable.

We briefly introduced the structures inherent in software and discussed the applicability of three graph models, namely the call graph, the simple reference graph and the hierarchic reference graph. Based on the last model, we elaborated the requirements for a general visualization of software and defined the layout problem we intended to solve. Our goal was to draw a layout in which the hierarchy of a graph is emphasized since it is the part of the software structure that usually reflects the design intended by the developer. We also considered that we wanted an interactive and fully scalable map; by using the hierarchy we can support features like zoom and the clarity of the map is improved with the help of intelligent edge bundling.

Next, we analyzed the capabilities of different layout methods. We selected force-directed techniques because they can easily be adjusted to special needs. We proposed an algorithm that is able to compute a suitable layout in several smaller steps. Each vertex is processed independently to generate a partial layout for its children disregarding external vertices. Since the partial layouts cover only one level of the hierarchy, this strongly reduces the complexity of our original problem. By handling the vertices in bottom-up order, we can incrementally assemble the global layout of the graph. Furthermore, we isolated the vertex placement and the edge routing in partial layouts. Vertices are placed by a modified Fruchterman-Reingold algorithm that can deal with two-dimensional vertices. For the bundling of edges, we again used the hierarchy of the graph and the edge routing is computed through a simple geometric heuristic.

Afterwards, we examined the effects of the parameters of the algorithm on the layout and explained them with practical examples. In particular, we rated the importance of the specific parameters and derived instructions on how to choose a proper parameter set. We also presented our results, pointed out deficiencies and suggested how to build on our visualization.

To summarize, we are able to produce nice layouts that facilitate the exploration of software projects. However, we found significant problems in rendering the graphs in real time: These issues have to be solved if we want to create the corresponding interactive visualization tools. Besides, the algorithm we designed is effective, but not very efficient and could be enhanced in terms of performance as well as quality.

Bibliography

[1] Ala Abuthawabeh, Fabian Beck, Dirk Zeckzer, and Stephan Diehl. “Finding structures in multi-type code couplings with node-link and matrix visualizations”. In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*. 2013, pp. 1–10. DOI: 10.1109/VISSOFT.2013.6650530.

[2] Craig Anslow, Stuart Marshall, and James Noble. “X3D-Earth in the Software Visualization Pipeline”. In: *Proceedings of the X3D-Earth Technical Requirements Workshop, Naval Postgraduate School, Monterey, California, USA, November 14-15, 2006*. 2006.

[3] Thomas Ball and Stephen G. Eick. “Software Visualization in the Large”. In: *IEEE Computer* 29.4 (1996), pp. 33–43. DOI: 10.1109/2.488299.

[4] Gergo Balogh and Árpád Beszédes. “CodeMetropolis - A minecraft based collaboration tool for developers”. In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*. 2013, pp. 1–4. DOI: 10.1109/VISSOFT.2013.6650528.

[5] Michael Balzer, Oliver Deussen, and Claus Lewerentz. “Voronoi treemaps for the visualization of software metrics”. In: *Proceedings of the ACM 2005 Symposium on Software Visualization, St. Louis, Missouri, USA, May 14-15, 2005*. 2005, pp. 165–172. DOI: 10.1145/1056018.1056041.

[6] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. “Software Landscapes: Visualizing the Structure of Large Software Systems”. In: *VisSym 2004, Symposium on Visualization, Konstanz, Germany, May 19-21, 2004*. 2004, pp. 261–266.

[7] Josh Barnes and Piet Hut. “A hierarchical O(N log N) force-calculation algorithm”. In: *Nature* 324.6096 (1986), pp. 446–449. DOI: 10.1038/324446a0.

[8] Michael Baur and Ulrik Brandes. “Multi-circular Layout of Micro/Macro Graphs”. In: *Graph Drawing*. Ed. by Seok-Hee Hong, Takao Nishizeki, and Wu Quan. Vol. 4875. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 255–267. DOI: 10.1007/978-3-540-77537-9_26.

[9] Jon L. Bentley and Thomas Ottmann. “Algorithms for Reporting and Counting Geometric Intersections”. In: *IEEE Transactions on Computers* 28.9 (1979), pp. 643–647. DOI: 10.1109/TC.1979.1675432.

[10] Dirk Beyer. “Co-Change Visualization”. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume, ICSM 2005, 25-30 September 2005, Budapest, Hungary*. 2005, pp. 89–92.

[11] Christian S. Collberg, Stephen G. Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. "A System for Graph-Based Visualization of the Evolution of Software". In: *Proceedings ACM 2003 Symposium on Software Visualization, San Diego, California, USA, June 11-13, 2003.* 2003, pp. 77–86, 212–213.

[12] Marco D'Ambros and Michele Lanza. "Software Bugs and Evolution: A Visual Approach to Uncover Their Relationship". In: *10th European Conference on Software Maintenance and Reengineering (CSMR 2006), 22-24 March 2006, Bari, Italy.* 2006, pp. 229–238. doi: 10.1109/CSMR.2006.51.

[13] Lionel E. Deimel, Jr. "The Uses of Program Reading". In: *SIGCSE Bulletin* 17.2 (1985), pp. 5–14. doi: 10.1145/382204.382524.

[14] David P. Dobkin, Emden R. Gansner, Eleftherios Koutsofios, and Stephen C. North. "Implementing a General-Purpose Edge Router". In: *Graph Drawing, 5th International Symposium, GD '97, Rome, Italy, September 18-20, 1997, Proceedings.* 1997, pp. 262–271. doi: 10.1007/3-540-63938-1_68.

[15] Ugur Dogrusoz, Erhan Giral, Ahmet Cetintas, Ali Civril, and Emek Demir. "A layout algorithm for undirected compound graphs". In: *Information Sciences* 179.7 (2009), pp. 980–994. doi: 10.1016/j.ins.2008.11.017.

[16] Tim Dwyer, Kim Marriott, and Michael Wybrow. "Integrating Edge Routing into Force-Directed Layout". In: *Graph Drawing, 14th International Symposium, GD 2006, Karlsruhe, Germany, September 18-20, 2006. Revised Papers.* 2006, pp. 8–19. doi: 10.1007/978-3-540-70904-6_3.

[17] Tim Dwyer and Lev Nachmanson. "Fast Edge-Routing for Large Graphs". In: *Graph Drawing, 17th International Symposium, GD 2009, Chicago, IL, USA, September 22-25, 2009. Revised Papers.* 2009, pp. 147–158. doi: 10.1007/978-3-642-11805-0_15.

[18] Peter Eades. "Drawing Free Trees". In: *Bulletin of the Institute of Combinatorics and its Applications* 5 (1992), pp. 10–36.

[19] Peter Eades and Qing-Wen Feng. "Multilevel Visualization of Clustered Graphs". In: *Graph Drawing, Symposium on Graph Drawing, GD '96, Berkeley, California, USA, September 18-20, Proceedings.* 1996, pp. 101–112. doi: 10.1007/3-540-62495-3_41.

[20] Ozan Ersoy, Christophe Hurter, Fernando Vieira Paulovich, Gabriel Cantareiro, and Alexandru Telea. "Skeleton-Based Edge Bundling for Graph Visualization". In: *IEEE Trans. Vis. Comput. Graph.* 17.12 (2011), pp. 2364–2373. doi: 10.1109/TVCG.2011.233.

[21] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. "Live trace visualization for comprehending large software landscapes: The ExplorViz approach". In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013.* 2013, pp. 1–4. doi: 10.1109/VISSOFT.2013.6650536.

[22] Eclipse Foundation. *Eclipse JDT*. 2015. URL: `https://eclipse.org/jdt/`.

[23] Thomas M. J. Fruchterman and Edward M. Reingold. "Graph Drawing by Force-directed Placement". In: *Software: Practice and Experience* 21.11 (1991), pp. 1129–1164. DOI: `10.1002/spe.4380211102`.

[24] Pawel Gajer and Stephen G. Kobourov. "GRIP: Graph Drawing with Intelligent Placement". In: *Graph Drawing*. Ed. by Joe Marks. Vol. 1984. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 222–228. DOI: `10.1007/3-540-44541-2_21`.

[25] Emden R. Gansner, Yifan Hu, Stephen C. North, and Carlos E. Scheidegger. "Multilevel agglomerative edge bundling for visualizing large graphs". In: *IEEE Pacific Visualization Symposium, PacificVis 2011, Hong Kong, China, March 1-4, 2011*. 2011, pp. 187–194. DOI: `10.1109/PACIFICVIS.2011.5742389`.

[26] Michael T. Gastner and Mark E. J. Newman. "Diffusion-based method for producing density-equalizing maps". In: *Proceedings of the National Academy of Sciences of the United States of America* 101.20 (2004), pp. 7499–7504. DOI: `10.1073/pnas.0400280101`. eprint: `http://www.pnas.org/content/101/20/7499.full.pdf+html`.

[27] Daniel M. Germán. "An empirical study of fine-grained software modifications". In: *Empirical Software Engineering* 11.3 (2006), pp. 369–393. DOI: `10.1007/s10664-006-9004-6`.

[28] Carlos Gouveia, José Campos, and Rui Abreu. "Using HTML5 visualizations in software fault localization". In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*. 2013, pp. 1–10. DOI: `10.1109/VISSOFT.2013.6650539`.

[29] Nuzhat J. Haneef. "Software Documentation and Readability: A Proposed Process Improvement". In: *ACM SIGSOFT Software Engineering Notes* 23.3 (1998), pp. 75–77. DOI: `10.1145/279437.279470`.

[30] Frank Harary. *Graph Theory*. Addison-Wesley, 1969.

[31] Joshua W. K. Ho and Seok-Hee Hong. "Drawing Clustered Graphs in Three Dimensions". In: *Graph Drawing, 13th International Symposium, GD 2005, Limerick, Ireland, September 12-14, 2005, Revised Papers*. 2005, pp. 492–502. DOI: `10.1007/11618058_44`.

[32] Danny Holten. "Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data". In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (2006), pp. 741–748. DOI: `10.1109/TVCG.2006.147`.

[33] Danny Holten and Jarke J. van Wijk. "Force-Directed Edge Bundling for Graph Visualization". In: *Computer Graphics Forum* 28.3 (2009), pp. 983–990. DOI: `10.1111/j.1467-8659.2009.01450.x`.

[34] Stephen G. Kobourov. "Spring Embedders and Force Directed Graph Drawing Algorithms". In: *CoRR* abs/1201.3011 (2012).

[35] Joseph B. Kruskal. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem". In: *Proceedings of the American Mathematical Society* 7.1 (1956), pp. 48–50.

[36] Adrian Kuhn, David Erni, Peter Loretan, and Oscar Nierstrasz. "Software Cartography: thematic software visualization with consistent layout". In: *Journal of Software Maintenance* 22.3 (2010), pp. 191–210. doi: `10.1002/smr.414`.

[37] Ming C. Lin and Stefan Gottschalk. "Collision detection between geometric models: a survey". In: *Proceedings of the IMA Conference on Mathematics of Surfaces*. Vol. 1. 1998, pp. 602–608.

[38] Vijay Krishna Palepu and James A. Jones. "Visualizing constituent behaviors within executions". In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*. 2013, pp. 1–4. doi: `10.1109/VISSOFT.2013.6650537`.

[39] Thomas Panas, Rebecca Berrigan, and John C. Grundy. "A 3D Metaphor for Software Production Visualization". In: *Seventh International Conference on Information Visualization, IV 2003, 16-18 July 2003, London, UK*. 2003, pp. 314–319. doi: `10.1109/IV.2003.1217996`.

[40] Sergey Pupyrev, Lev Nachmanson, Sergey Bereg, and Alexander E. Holroyd. "Edge Routing with Ordered Bundles". In: *Graph Drawing - 19th International Symposium, GD 2011, Eindhoven, The Netherlands, September 21-23, 2011, Revised Selected Papers*. 2011, pp. 136–147. doi: `10.1007/978-3-642-25878-7_14`.

[41] Darrell R. Raymond. "Reading source code". In: *Proceedings of the 1991 Conference of the Centre for Advanced Studies on Collaborative Research, October 28-30, 1991, Toronto, Ontario, Canada*. 1991, pp. 3–16. doi: `10.1145/962113`.

[42] Steven P. Reiss and Alexander Tarvo. "Tool Demonstration: The Visualizations of Code Bubbles". In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*. 2013, pp. 1–4. doi: `10.1109/VISSOFT.2013.6650521`.

[43] Frank Wagner and Alexander Wolff. "A Practical Map Labeling Algorithm". In: *Computational Geometry* 7.5-6 (1997), pp. 387–404. doi: `10.1016/S0925-7721(96)00007-7`.

[44] Jan Waller, Christian Wulf, Florian Fittkau, Philipp Dohring, and Wilhelm Hasselbring. "Synchrovis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency". In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*. 2013, pp. 1–4. doi: `10.1109/VISSOFT.2013.6650520`.

[45] Richard Wettel. "Software Systems as Cities". PhD thesis. Università della Svizzera Italiana, Sept. 2010.

[46] Richard Wettel and Michele Lanza. “Visualizing Software Systems as Cities”. In: *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2007, 25-26 June 2007, Banff, Alberta, Canada*. 2007, pp. 92–99. doi: 10.1109/VISSOF.2007.4290706.

[47] Kai Xu, Andrew Cunningham, Seok-Hee Hong, and Bruce H. Thomas. “GraphScape: Integrated Multivariate Network Visualization”. In: *APVIS 2007, 6th International Asia-Pacific Symposium on Visualization 2007, Sydney, Australia, 5-7 February 2007*. 2007, pp. 33–40. doi: 10.1109/APVIS.2007.329306.

[48] Peter Young and Malcolm Munro. “Visualising Software in Virtual Reality”. In: *6th International Workshop on Program Comprehension (IWPC ’98), June 24-26, 1998, Ischia, Italy*. 1998, pp. 19–26. doi: 10.1109/WPC.1998.693276.