FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

CleanGraph II - Improved Graph Drawings of Family Trees and UML Class Diagrams

Moisés Pimenta da Rocha

Mestrado em Engenharia Informática e Computação

Supervisor: Daniel Castro Silva

July 30, 2022

CleanGraph II - Improved Graph Drawings of Family Trees and UML Class Diagrams

Moisés Pimenta da Rocha

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Rui Rodrigues

External Examiner: Prof. João Jacob

Supervisor: Prof. Daniel Silva

July 30, 2022

Abstract

Information visualization is an important aspect in the lives of many people and how they perform their tasks. One of the way certain types of data can be represented is through the use of graphs, which model various structures. One of the main challenges of representing graphs is the ever increasing quantity and complexity of data. In a previous work, a tool called CleanGraph was developed to address certain issues in current visualization and interaction tools for Family Trees and Universal Modelling Language (UML) class diagrams.

CleanGraph differentiates itself with the capability for full graph representation along with new interaction features that allow for a productivity boost, in certain tasks, over existing tools. However, much work can still be done: the implemented layout algorithms don't make an efficient use of space, and the resulting layouts are quite static.

In order to improve this tool, two main tasks are devised: (1) research and implement state of the art graph layout algorithms, the research has been done, for Family Trees many methods were found mostly relying of partial layout, for UML class diagrams two general methods were found that produce satisfactory layouts for diagrams with specific properties; (2) improve the graph interaction features with graph animations and secondary layouts.

For Family Trees a hierarchical approach using a general purpose algorithm was used. To make this possible the graph representation used in CleanGraph was modified to be compatible with the algorithm. Further changes were needed to accommodate this modification across the tool.

For UML Class Diagrams, a Topology-Shape-Metrics approach was used. Some modifications to the graph representation of UML class diagrams used by CleanGraph were also needed to convert it into a proper graph, so the algorithm could operate over it. There is also further work on the radial layout feature of CleanGraph, taking inspiration from MoireGraphs.

Upon comparison between the old and new solutions and analysis of the new one, there are possible pros and cons in the new layout for Family Trees. There are also a few extension opportunities that will allow the new layout to become more aesthetically pleasing.

In the case of UML diagrams, there is a clear improvement over the previous solution in the resultant drawings, namely the avoidance of edge and node overlap. The new layout also achieves a orthogonal aesthetic which is common in UML diagrams. However, some improvement still need to be made, namely handling of certain special cases that the previous solution handled by default.

In conclusion, there was a significant improvement in CleanGraph's capabilities to display Family Trees and UML class diagrams in certain aspects. But, these improvements come at a cost in other areas, leaving some future work still to be done, to make CleanGraph an even better tool for users seeking improved methods to view their ancestry data, as well as software architects, project managers and system analysts, who would benefit from a better platform for system

representation.

Keywords: Diagrams, Interaction, Unified Modelling Language, Family Trees, Genealogical Graphs, Graph Layout, Information Visualization

Contents

List of Figures

List of Tables

Abbreviations

BFS	Breath-First Search
DAG	Directed Acyclic Graph
DFS	Depth-First Search
GEDCOM	Genealogical Data Communications
InfoVis	Information Visualization
NASA	National Aeronautics and Space Administration
OMG	Object Management Group
UML	Unified Modelling Language

Chapter 1

Introduction

Information visualization has existed ever since humans needed to share knowledge about a subject, and this can date back to prehistoric times, when humans engraved surfaces to communicate experiences [Friendly and Wainer, 2021]. One of the many ways to represent information is through graphs as their relational nature allows for the modeling of natural and human structures and, through the use of graph theory, solving complex real-world practical problems [Riaz and Ali, 2011].

Many individuals' functions in society require the correct interpretation of data, be it politicians, data analysts, or consultants. This interpretation can be done through the use of statistics, which can be used to describe large samples of raw data, as raw data can be hard or impossible to analyze. Another way to interpret data can be through the use of diagrams, which allow for a more visual approach. However, diagrams must be properly designed and communicated to transmit the intended information [Purchase, 2014].

Information Visualisation (InfoVis) Systems are systems that allow users to interact with visualizations of data to extract understanding from it. They differentiate themselves from information graphics as these only present static visual representations of data. InfoVis Systems leverage visual, textual, and interactive elements to create different views of data [Sorapure, 2019].

1.1 Context and Motivation

CleanGraph was developed in a previous dissertation work by Martins [Martins, 2021] with the intent of addressing some shortcomings in Family Tree and UML class diagram visualization tools. Of the implemented features, some aspects of the tool can be noted. The tool does full graph representation (see Fig 1.1), which in the case of family trees, is a differentiating factor from many solutions currently on the market. Pairing that up with new ways of data presentation (see Fig 1.2) and exploration resulted in a tool that registered an overall increase in productivity in certain tasks against common tools.

However, the final product still leaves some work to be done. Firstly, the majority of the work done on the tool focused more on the Family Tree representation and interaction side. The

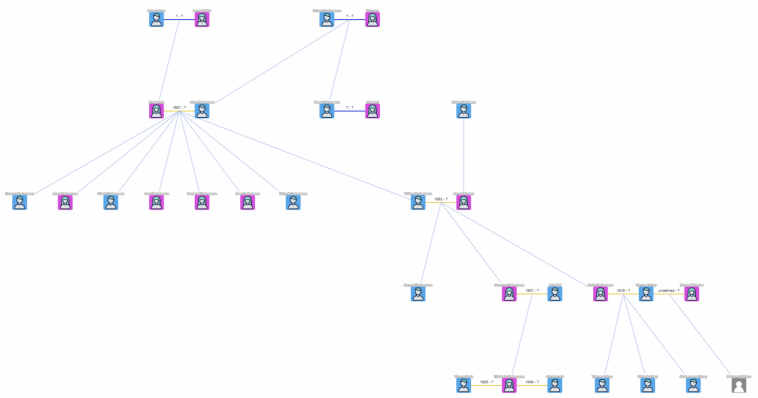


Figure 1.1: Example of a full family tree representation in CleanGraph [Martins, 2021].

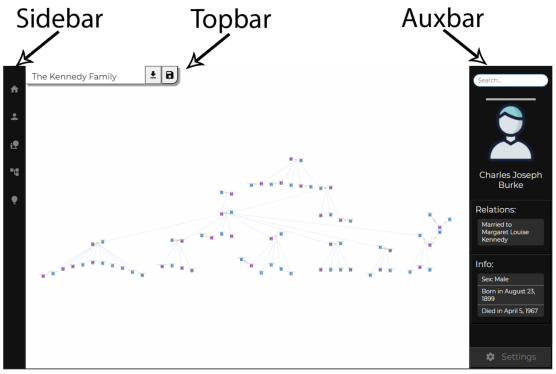
tool's UML class diagram representation and interaction side was worked on but not developed as much, which can be reflected by the lack of testing done to that section. This bias is because a significant part of the project included building an interaction model and interface for the tool, the development of a back-end server, which had parsing libraries to ingest specification files to generate the graphs, and the development of graph layout algorithms for each of the types of diagram. With this much work constrained by the dissertation time window, it was inevitable that some parts would be left underdeveloped.

Of these parts, the notable ones are the implemented graph layout algorithms. While these algorithms appear to give clean, readable drawings for small to medium graphs, their shortcomings become apparent when the graph size increases further. One of the main issues is the inefficient use of space, especially on Family Trees. The present layouts leave much white space that could be better utilized (see Fig 1.3). Conveniently, the datasets used in the examples render good, readable layouts. However, when the scale starts to increase, it is expected that edge crossings and even edges overlapping with nodes begin to appear. Another point of improvement is the fact that the layouts are relatively static. This opens a door for the improvement of the interaction aspect. Although CleanGraph does have features that allow interaction with the layout, like zooming, panning, and dragging nodes. The tool could also have more dynamic layouts that rearrange themselves as the user navigation the data.

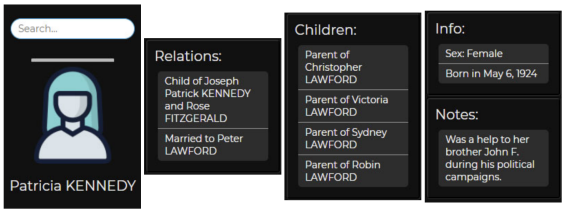
1.2 Objectives

This work aims to improve the graph visualizations of the tool, making better use of the space while still providing good, readable layouts of Family Trees and UML class diagrams, as well as improve graph exploration and navigation through interactions with the actual graph. These can rely on many commonly used interaction techniques like animated transitions.

To improve the graph visualization and interaction of CleanGraph new graph layout algorithms will be implemented to represent Family Trees and UML class diagrams, and new interaction



(a) Overall interface.



(b) Sections of the auxbar presenting information about a person.

Figure 1.2: CleanGraph user interface [Martins, 2021].

methods will also be implemented. These tasks will be performed while attempting to answer the following questions:

- **Q1**: How can we achieve more space efficient and readable Family Tree and UML class diagram visualizations?
- **Q2**: Can animated graph navigation boost graph exploration productivity?

Answers to these questions are obtained through testing the work done and checking whether the results match the expected outcomes. For **Q1**, the testing can be related to measuring specific aesthetic characteristics that directly correlate to a graph's readability, e.g., number of edge crossings, edge lengths, number of edge bends, etc. For **Q2**, tests similar to the ones in the previous work can be done [Martins, 2021]. Having people with experience with dealing with these kinds of diagrams go through a set of tasks while measuring the time to complete each one, and the rate of mistakes can provide a measure of the productivity enabled by the platform. In addition, questionnaires to extract points about the user experience may also help in this manner.

To ensure proper results, some goals can be defined to guide the development of the improvements [Victorelli et al., 2020] [Locoro et al., 2017], some are inherited from the previous work [Martins, 2021]:

- **Data Representation Accuracy** — The graphs and respective layouts that are created must follow the rules set by the underlying data type. If no rules are specified, the representation should try and be as informative and clear as possible.

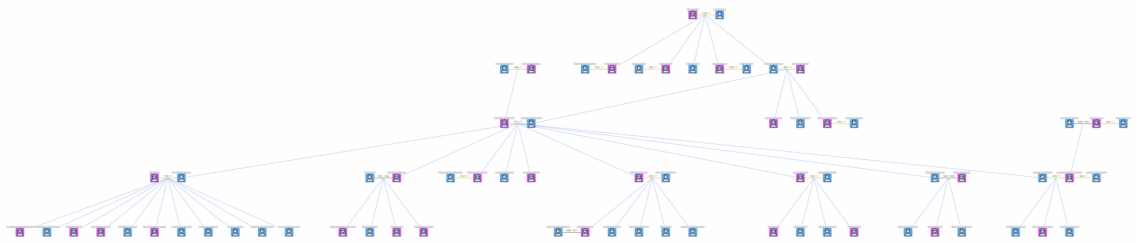


Figure 1.3: Layout of a considerably bigger tree in CleanGraph, inefficient use of space starts to become apparent [Martins, 2021].

- **Readability** — The produced layouts should be easy to understand, with minimal clutter and good aesthetics.
- **Intuitivity** — The interaction methods to be implemented need to be capable of organizing information in such a way that a user can identify certain properties quickly.
- **Usability** — This work seeks an improvement of CleanGraph's current graph visualization and navigation. So, the results need to provide a user experience that is comparable or better than the one provided by the current CleanGraph version and other solutions.

The main contributions of this work are the new layout algorithm implementations that hope to improve the visualization of both Family Trees and UML class diagrams in CleanGraph. It also provides discussions on how these algorithms can be extended to improve the drawings further or support more features or special cases.

1.3 Dissertation Structure

Along with the introduction this document features three more chapters.

Chapter 2 introduces Family Trees and UML class diagrams, some history of both, with visual aspects. Ways of how these can be represented as graphs are also introduced.

In Chapter 3, the related work is discussed. This includes works on graph layout methods for graphs in general and more specialized for Family Trees and UML class diagrams. As well as interaction features that are bundled with these layouts.

In Chapter 4, problems regarding graph layout are discussed and related with the specific constraints that might be introduced with Family Trees and UML class diagrams. It also addresses how new interaction methods may be implemented in CleanGraph and a decision is made on certain aspects of each layout as well as the additional interaction feature to be implemented.

In Chapter 5 goes over the implementations of the chosen algorithms for the layout of the graphs, as well as the implementations of interaction features. Implementation details like graph representation modifications are addressed here as well.

Chapter 6 presents comparisons between the results of the new layouts and the old ones. Some pros as cons are drawn and possible extensions to the algorithms are presented.

Chapter 7 presents conclusions and future work.

Chapter 2

Background

This chapter introduces some graph concepts that will be useful to understand throughout this dissertation. It also introduces Family Trees and UML class diagrams and how they can be related to graphs. The latter is especially relevant to understanding how the state of the art works over them. A short overview of CleanGraph's core features is also done.

2.1 Core Graph Concepts

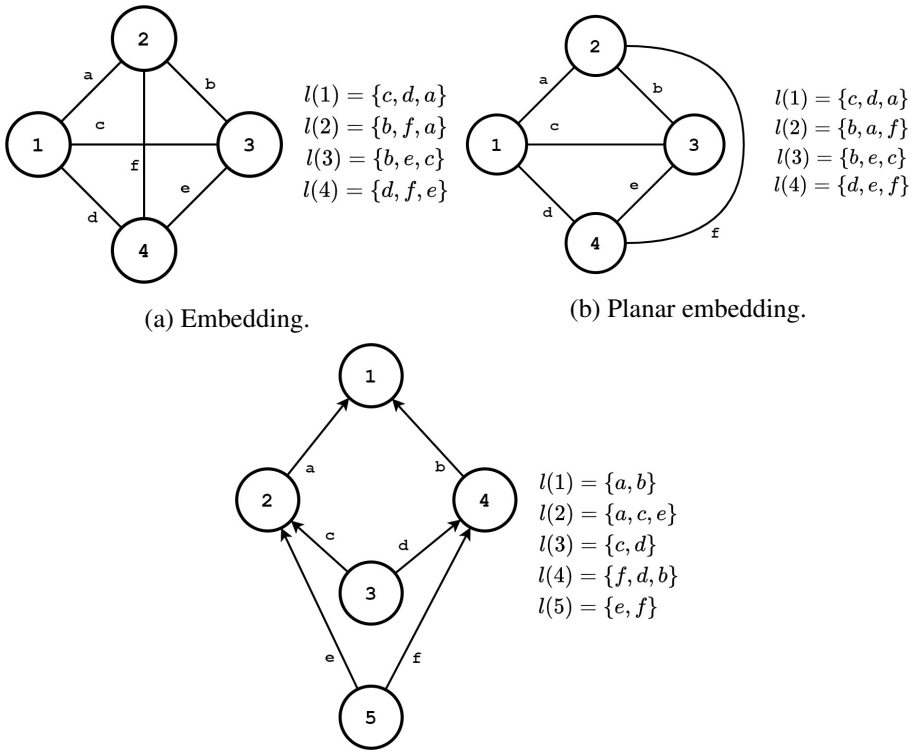
This section will cover and explain some core graph theory concepts that will be useful in understanding some of the algorithms that will be covered in this dissertation. They are also important in understanding some parts of the implementation chapter as well as the ideas suggested in the discussion chapter.

2.1.0.1 Graph Embedding

An embedding of a graph can be regarded as an equivalence class of drawings of that graph. It associates each vertex v of the graph to a list $l(v)$ of its incident edges sorted in the clockwise order that they appear around said vertex [Eiglsperger and Kaufmann, 2001] [Bertolazzi et al., 1994]. Figure 2.1a shows an example of an embedding. It is to note that the initial element isn't relevant. What matters is that the edges are sorted based on their clockwise order around the node.

A graph embedding is considered planar if there is a drawing of the graph that preserves that embedding and contains no edge crossings [Eiglsperger and Kaufmann, 2001], i.e., edges intersect only at their endpoints. The edges in the drawing don't need to be straight lines. Figure 2.1b shows a planar embedding of the graph from Fig. 2.1a along with a corresponding drawing.

In directed graphs, embeddings can be further constrained to upward planar embeddings. An embedding of a graph is upward planar if there is a drawing that preserves the embedding in which all edges increase monotonically in the upward direction [Eiglsperger and Kaufmann, 2001]. Figure 2.1c shows an example of such an embedding along with a corresponding drawing. It is implied that for such a condition to be possible, the graph needs to be acyclic.



(a) Embedding.

(b) Planar embedding.

(c) Upwards planar embedding.

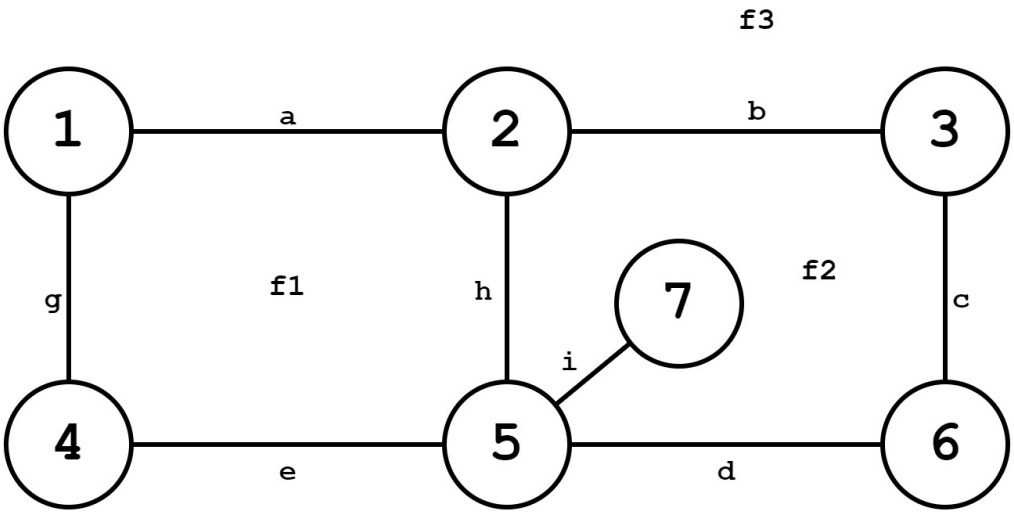
Figure 2.1: Embedding examples.

An interesting characteristic of upwards planar embeddings is that the incoming and outgoing edges form non-overlapping ranges in the planar embeddings. This also means that, in drawings of upwards planar embeddings, if, for each vertex, a horizontal line is drawn through its center, the vertex's incoming edges will be situated below this line, and outgoing edges will be situated above it. Every edge in the graph will have an angle α with the horizontal in the range $0° < \alpha < 180°$ [Bertolazzi et al., 1994].

Finally, there are mixed upward planar embeddings. An embedding of a mixed graph, i.e. a graph with both directed and undirected edges, is mixed upward planar if there is a drawing that preserves the embedding and in which the directed edges in the graph are represented by arcs monotonically increasing in the vertical direction [Eiglsperger and Kaufmann, 2001]. Undirected edges can take whichever path.

2.1.0.2 Graph Faces

When a graph is drawn without edge crossings, it divides the plane into a set of regions, called the faces of a graph [Bertolazzi et al., 1994]. Generally, they are defined by a clockwise circular ordering of the edges that define the region's boundary. Figure 2.2 shows an example of a planar graph drawing with the definition of its faces. Relevant things to point out are that edges appear exactly two times across the definition of all faces; it is possible for both times to be on the same face, as exemplified with face $f2$ in Fig. 2.2. By convention, the unbounded region outside of the



$$f1 = \{a, h, e, g\}$$
$$f2 = \{b, c, d, i, i, h\}$$
$$f3 = \{b, a, g, e, d, c\}$$

Figure 2.2: Example of a planar graph and the definition of it's faces.

graph is also considered a face, and its definition has edges in the reverse order, as shown with face $f3$. This face is generally referred to as the outer or external face.

The faces of a graph can be calculated algorithimically, granted a planar embedding of the graph is available. Given a graph $G = (V, E)$ with a set of nodes V and a set of edges E, and a planar embedding of G that maps every node $v \in V$ to a list $l(v)$ containing a clockwise ordering of its incident edges, the algorithm to calculate the graph's faces goes as follows:

1. Pick the node $v \in V$ with the lowest degree which hasn't been fully processed.
2. Pick an incident edge $e = (v, w)$ that hasn't been visited yet.
3. In w pick the edge j that sits just before edge e in $l(w)$, i.e. the next edge in the counter-clockwise order.
4. Repeat step 3 until v is reached once again.

In the context of node v, an edge is considered visited if it was traversed in the direction exiting node v. A node is considered fully processed if all of its incident edges were visited. Figure 2.3 provides a visualization of the order of the edge traversal for each face in the graph from Fig. 2.2.

2.1.0.3 Orthogonal Representation

An Orthogonal Representation H of a graph G describes the generic orthogonal shape of the faces of G, being very useful for creating orthogonal drawings of graphs [Tamassia, 1987]. It is a

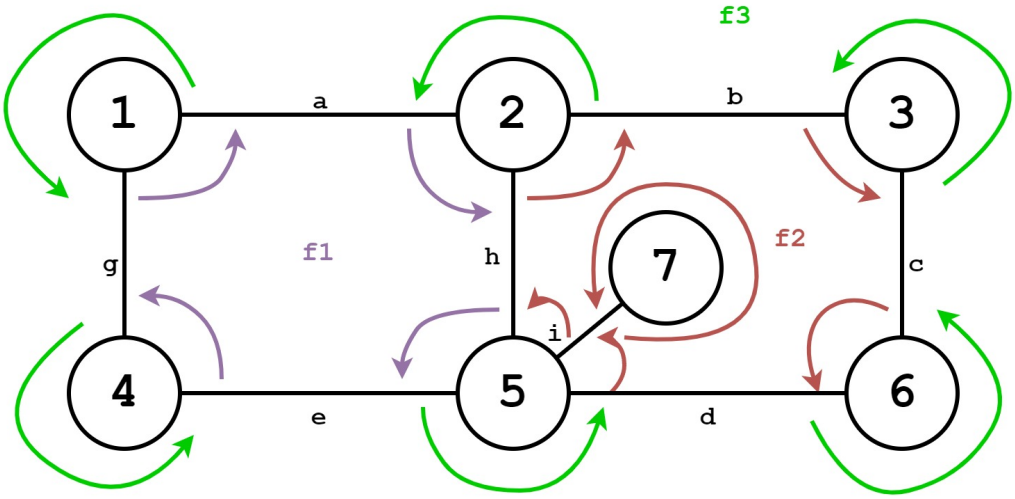
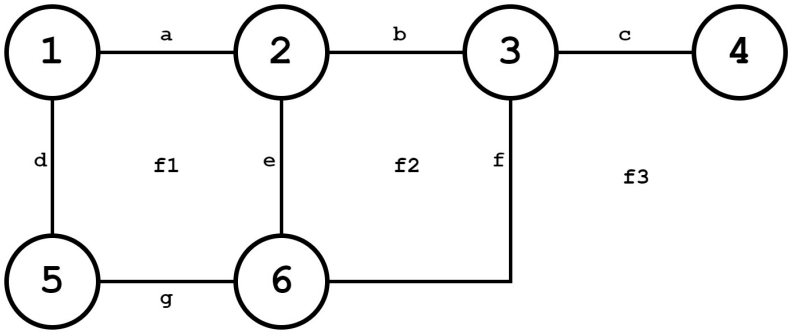


Figure 2.3: Graph from Fig. 2.2 with edge traversal orders.



$$H(f1) = \{(a,1,\{\}),(e,1,\{\}),(g,1,\{\}),(d,1,\{\})\}$$
$$H(f2) = \{(b,1,\{\}),(f,1,\{0\}),(e,1,\{\})\}$$
$$H(f3) = \{(a,3,\{\}),(d,3,\{\}),(g,2,\{\}),(f,1,\{1\}),(c,4,\{\}),(c,2,\{\}),(b,2,\{\})\}$$

Figure 2.4: Orthogonal drawing of a graph with accompanying orthogonal representation.

mapping from the set of faces F of the graph to lists of tuples (e_r, a_r, b_r), where e_r is an edge, a_r is the angle e_r makes with the following edge, stored as a multiple of 90°, and b_r a list of bends of the edge. In each tuple, $a_r \in \mathbb{Z}$ is constrained to $1 <= a_r <= 4$, and b_r is a list of ones and zeros, ones represent angles of 270° and zeros represent angles of 90°. Figure 2.4 shows an example of such a representation.

Since a_r is constrained to being at least 1, it is implied that for there to be a possible orthogonal representation of a graph, each of its nodes cannot have more than four incident edges, meaning as well that each edge is assigned to a unique side of the node. For graphs where nodes have more than four incident edges, a Quasi-Orthogonal Representation is required. It is defined in the same way as Orthogonal Representations with the addition that it allows a_r to be 0 as well, i.e. 0° angles are allowed between edges, which means that multiple edges can be attached to the same side of a node.

2.2 Family Trees

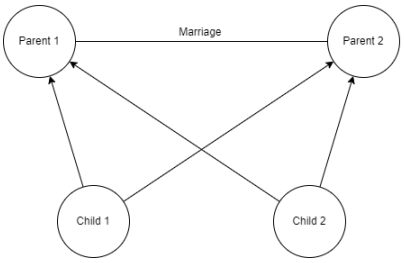
Family Trees are charts that represent people and their family relationships [Collins English Dictionary, 2021]. The most common visual representation of Family Trees might just be a hierarchical tree-like structure, as is evidenced in a state-of-the-art study in [Martins, 2021].

Family trees can also be referred to as pedigree charts or genealogies and can be useful to track a person's ancestors. This is a focal part of genealogy [Sharpe, 2011]. This type of work has been important since ancient times. Back then, a person's pedigree would usually define their social status. In ancient Greece, genealogy was used to prove descent from gods, which gave people social status. However, these methods weren't very scientific. In ancient Rome, genealogy was practiced to distinguish between the plebeians and people of noble descent. In the Old Testament, genealogies were used to provide myths, "helping to define Israel's nationhood, and confirming the authority of its kings and priests" [Sharpe, 2011]. In Medieval Europe, genealogies were also used in the politics of inheritance and succession [Pálsson, 2002].

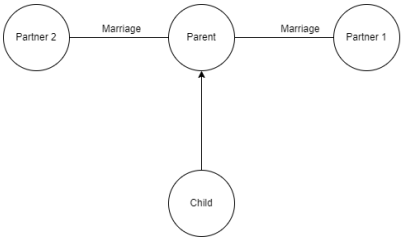
While family trees use the tree analogy in their representation, they might not be trees in the strict sense used in graph theory. Since it is possible for distant relatives to marry and have children, the path between an ancestor and descendant may not be unique anymore [McGuffin and Balakrishnan, 2005]. Trees also assume, in general, a single root node from which the entire structure proliferates. However, family trees may have more than one root node in the form of individuals with no ancestors. In this case, family trees are better considered as normal graphs, and since ancestors are always born before their children, these graphs are actually directed acyclic graphs (DAGs).

The visual representation of genealogical information as graphs can also be done in many different ways. These affect how the information is displayed. The most obvious approach would be to apply the definition of a graph, mapping people to nodes and relationships to edges (Fig. 2.5a). In a sense, this already represents the relationships very well on an information level. However, visualizing such a graph can be cumbersome. While it would be simple to identify

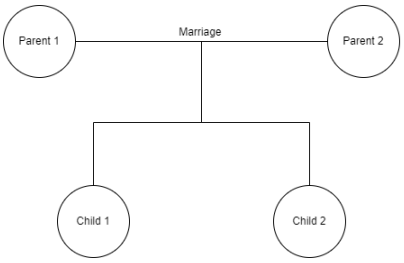
ancestors, descendants, and marriages, one of the characteristics of family trees is that it is possible to identify from which marriage a person was born (e.g. Fig. 2.5b shows a child whose parent had two marriages, but it is impossible to know which marriage the child originated from). Many solutions use notation such as or similar to the genogram notation to display such information, where married people are linked by an edge and their children are connected to the marriage edge and not their parents (Fig. 2.5c). CleanGraph also uses this approach to display family trees [Martins, 2021]. However, this notation does not represent a pure graph as there are no edges between edges and nodes in graph theory. An approach that maps family trees to graphs and allows for the information extraction mentioned above is to map certain relationships to nodes, such as marriages. Married people would then be connected to their respective marriage node, and the children resulting from such a marriage would also be connected to the respective node (Fig. 2.5d). The direction of the edges would then indicate who the parents are and who the children are. [Mařík, 2016] uses a representation like this to be able to apply a generic graph layout algorithm to draw family trees.



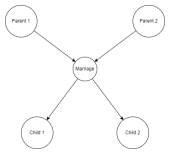
(a) Example of a direct mapping of genealogical information to a graph.



(b) Direct graph mapping of a family tree with 2 marriages.



(c) Representation of a marriage with two children through using a notation similar to genograms.



(d) Example of mapping of genealogical information to a graph where marriages are mapped to nodes.

Figure 2.5: Various representations of genealogical information as graphs.

2.3 UML Class Diagrams

The Unified Modelling Language (UML) is a language for modeling software systems. Its development was first done as an effort to unify the disparate modeling methods of Booch, Jacobson, and Rumbaugh, which were in turn developed in the 1990s to provide a modeling language in response to the needs of increasingly complex applications at the time[Booch et al., 1999]. The

three unified their method in an effort to stabilize the ecosystem, letting developers settle with a mature solution [Booch et al., 1999]. UML then became a language that lets users visualize, conceptualize and document their systems [Visual Paradigm team, 2021].

UML as a language is expressive enough to allow users to create different types of diagrams, each type useful to a different kind of stakeholder. These are divided into structure and behavior diagrams [OMG, 2017]. The types of diagrams this work is going to focus on are Class Diagrams. UML Class Diagrams are static diagrams that show the structure of a system by displaying its classes, along with its attributes, behavior, and relationships.

UML Class Diagrams are mainly comprised of class tables and relations connecting them. Class tables are divided into three compartments: the Title, printed in bold and centered, it is also capitalized; the Attributes, left aligned and lower camel case; the Operations, left aligned and lower camel case as well (Fig. 2.6).

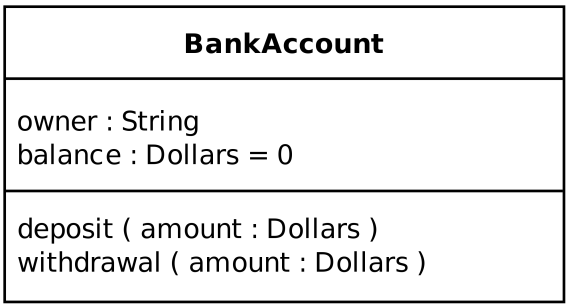


Figure 2.6: UML Class

Attributes and operations in a class can have a visibility, they must be prepended with the specific symbol: '**+**' for Public, '**-**' for Private and '**#**' for Protected

The scope of each member can also be defined: underlined members are working in the class scope.

Relations in UML are drawn as lines connecting the classes. The looks of the lines change depending on the type of relation. UML defines the following:

- **Generalization** — also called inheritance, it's a directional relation that indicates a super-class/sub-class relationship. It is drawn as a solid line with a single hollow white arrow pointing from the sub-class to the super-class. Usually, when there are multiple sub-classes to a super-class, the relations may converge into a single arrow (Figure 2.7a).

- **Dependency** — directed relation specifying a client/supplier relationship. In these relationships, a supplier has what a client needs for its implementation. It is represented as a dashed line with a thin arrow pointing from the client to the supplier (Figure 2.7b).

- **Association** — represents a link between two or more classes, can be directed or undirected and adorned with additional data such as names and multiplicities. Directed associations have a thin arrow at the end (Figure 2.7c).

- **Aggregation** — a kind of association that represent a "has a" relationship and is strictly binary. Aggregations can occur when a class serves as a container of instances of another class, but these instances don't rely on the container to exist, so, the container can be destroyed and the contained instances will still exist. It is represented as a solid line with a hollow diamond on the side of the container class (Figure 2.7d).
- **Composition** — similar to the Aggregation relation, however, with this relationship, if the container is destroyed, the contained instances also cease to exist. It is represented as a solid line with a filled diamond pointing towards the container class (Figure 2.7e).

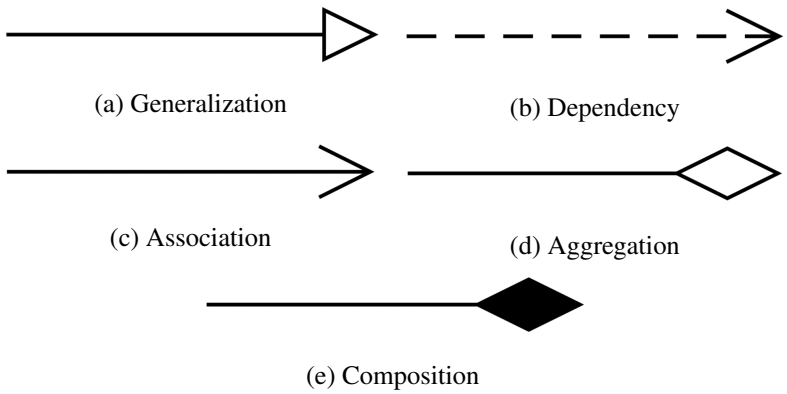


Figure 2.7: UML Relation Notations

Due to their nature, UML Class Diagrams end up resembling graphs, with classes being nodes and relations being edges. This becomes useful for researchers, as they can use various graph theory concepts to work over these diagrams, e.g. tackle the planar embedding problem to automatically draw diagrams, use clustering algorithms to automatically deduce modules in a system's architecture, etc. The benefits of these approaches become more evident once the scale of the diagrams increases as well. However, UML Class Diagrams can not always be represented as graphs due to certain elements. One such element is the Association Class. These are special classes that are used to label certain associations with more information. They have the appearance of normal classes and are connected to associations through an edge. Which in graph theory is not possible, as discussed above, since edges can't connect to other edges. So, adaptations are necessary. Chapter 5 covers how these situations are handled in the new CleanGraph layout algorithm implementations.

Definitions of UML Class Diagrams as graphs are very similar if not equal to the standard mathematical definition of a graph. A class diagram can be represented as a graph $G = (V, E)$ with a set of nodes V containing all classes and a set of edges E containing all relations between classes. Since class diagrams are very rich in information, these nodes and edges are then decorated with this information as attributes. The classes information doesn't affect how the graph will be represented. However, the relation information will affect the edges that represent these, namely the

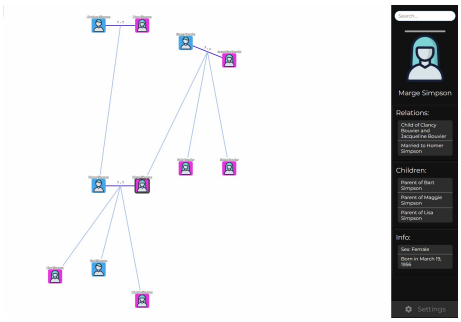


Figure 2.8: CleanGraph's interface.

directionality, e.g. associations may be bi or uni-directional, which means that edges may be directed or undirected. Uses of this representation can be found in many research works such as [Hu et al., 2012] and [Seemann, 1997] which define the graph in such a way, using the information stored in the attributes to guide the execution of their graph layout algorithms. [Eiglsperger et al., 2003] also uses this definition. However, prefers to separate the edges into different subsets as their algorithm only works with different kinds of edges at different stages. [Eiglsperger et al., 2004] also features an algorithm that represents the diagram as a graph $G = (V, A, E)$, where A are all edges representing generalizations and E are all other types of associations.

Lastly, class diagrams can also be defined as hypergraphs. These are generalizations of graphs in which edges join any number of vertices. This might be useful as generalization relations can be drawn as a merge of multiple arrows. In a hypergraph, this can be represented as an edge containing the super-class along with all of its sub-classes.

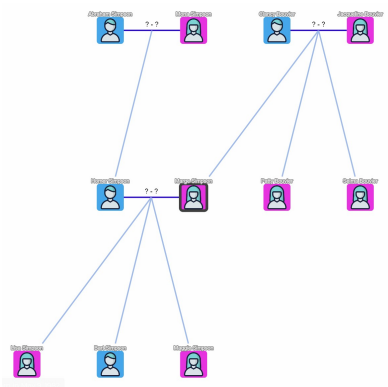
2.4 Clean Graph

This section will focus on introducing some of the core features in CleanGraph for the sake of building some prior knowledge of the platform [Martins, 2021]. This will help in understanding how the implemented feature in this work will compare with the pre-existing ones.

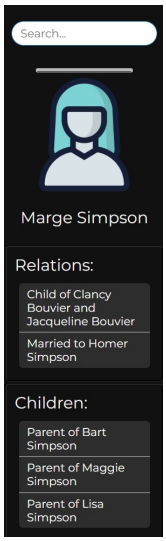
The interaction model of CleanGraph consists, at the basic level, of a viewport to navigate through graph visualizations and an auxiliary bar that sits to the right of the viewport (Fig. 2.8). The viewport for the navigation of graphs is provided by Cytoscape.js, which allows some form of interaction like zooming, panning, and dragging nodes around. The auxiliary bar to the right is used to display information about elements a user touches in the viewport. The auxiliary bar also contains a search bar to allow the user to search for elements in the graph. Which elements are searched for depends on the type of graph being displayed.

2.4.1 Family Trees

When a user loads a Family Tree file into CleanGraph, it will display a full layout of the family tree (e.g. Fig. 2.9a). This allows the user to quickly observe the entire Family Tree. The graph



(a) Full layout.



(b) Auxbar with person's information.

Figure 2.9: CleanGraph full layout of a Family Tree and auxiliary bar.

layout also has some information already visible, such as the name of the people in the Family Tree above their respective nodes.

Upon clicking on a person, the auxiliary bar will update to contain available information about that person, e.g. their birthday, their gender, their spouses, their children, etc (e.g. Fig 2.9b). Clicking on an edge will update the auxiliary bar to contain information about which type of relation the edge represents. The search bar in the auxiliary bar can be used to search for people whose name matches the input. The people that match are highlighted in the viewport.

The auxiliary bar in "Family Tree" mode contains a settings tab that allows for the manipulation of certain characteristics of the family tree visualization. Firstly, it allows toggling between the normal layout and a second layout that maps nodes to positions in the vertical axis according to their dates. This secondary layout is helpful in distinguishing the ages of people. Secondly, it has a function called *Generational Highlight*, which highlights people that are closely related to a selected person. How close they are related can be adjusted with a slider.

2.4.2 UML class diagrams

Like in Family Trees, when a UML class diagram is loaded into CleanGraph, a full graph layout will be computed. Clicking on a class will update the auxiliary bar with information about that class. When typing into the search bar to find classes, it doesn't just match the names of the classes but also attributes, operation names, and operation parameters. The classes that match the query are highlighted in the viewport.

The layout algorithm works by creating a radial style layout around a center class. Which class is at the center is determined automatically. However, the user can pick a new class to be

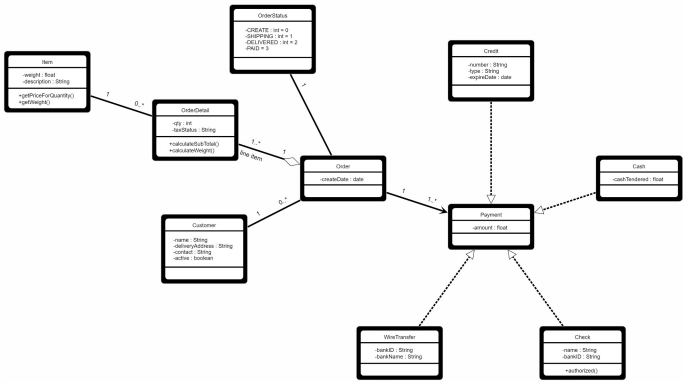


Figure 2.10: CleanGraph full UML class diagram layout.

the center class by double-clicking it. If the user does so, the layout rearranges itself around the selected class (e.g. Fig 2.11).

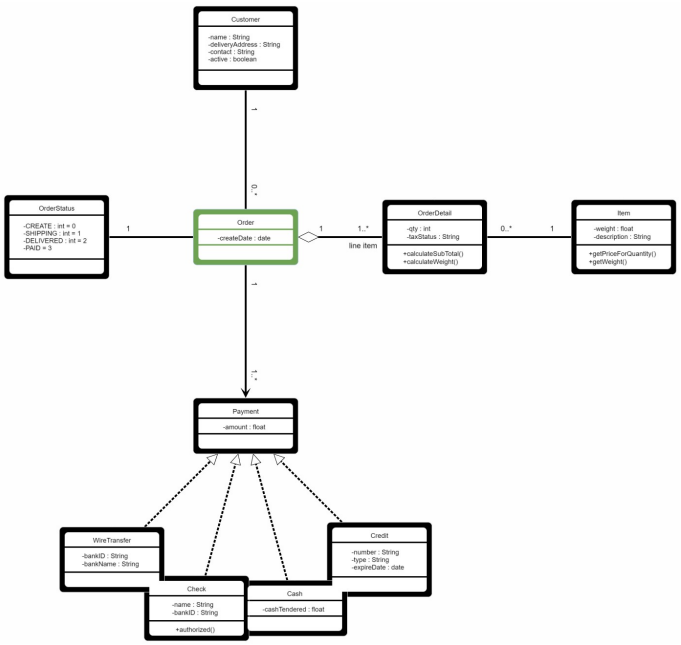


Figure 2.11: Rearranged layout around selected class.

Chapter 3

Related Work

This chapter dives into the state of the art in graph layout and interaction, and explores how these subjects are approached. Understanding is essential for the development of improvements to CleanGraph.

3.1 Graph Layout Algorithms

The first subject to study is automatic graph layout algorithms. Since one of the major objectives of this work is improving the current layouts in CleanGraph, understanding the state of the art in layout algorithms is essential. The ways in which graphs can be drawn are many and varied.

3.1.1 Generic Graph Layout Algorithms

Firstly, a look into more general-purpose algorithms is done. These can usually act as frameworks onto which further rules are applied to draw more specific data types.

3.1.1.1 Force-Directed Layouts

Force-directed graph layouts are achieved by modeling the graph as a physical system. Forces are assigned to nodes and edges based on their relative positions. These forces can then be used to simulate their movement or minimize the energy of the graph [Kobourov, 2012].

One of the more traditional force-directed methods is by Eades [Eades, 1984], where a mechanical model is used to layout simple graphs. The following quote succinctly explains the algorithm:

> ... we replace the vertices by steel rings and replace each edge with a spring to form a mechanical system ... The vertices are placed in some initial layout and let go so that the spring forces on the rings move the system to a minimal energy state.

Some adjustments were made to this initial model [Eades, 1984]: the springs have logarithmic strength due to the linear spring strength in Hooke's Law being too strong when vertices are far

apart, and non-adjacent vertices have a repelling force. This results in adjacent vertices being closer together and non-adjacent vertices being more apart.

Fruchterman and Reingold [Fruchterman and Reingold, 1991] further pioneer force directed graph layouts. The algorithm defines attractive forces that are applied to adjacent vertices and repelling forces that are applied to all pairs of vertices. This reflects their principles:

1. Vertices connected by an edge should be drawn near each other.
2. Vertices should not be drawn too close to each other.

It is similar to Eades' [Eades, 1984] approach as both define attractive forces for adjacent vertices and repelling ones for non-adjacent vertices. However, it further builds on the concept by adding a notion of "temperature" to the algorithm, which controls the displacement of vertices and becomes lower at every iteration, resulting in smaller adjustments as the graph approaches the ideal layout [Kobourov, 2012].

3.1.1.2 Hierarchical Layouts

Hierarchical layouts are a way of representing directed acyclic graphs (DAGs). These layouts divide nodes into layers (hierarchies) in such a way that all or almost all edges are pointing in the same direction.

Sugiyama et al. [Sugiyama et al., 1981] present a procedure to find hierarchical drawings of graphs. Nikolov sums up the framework with the following problem definition [Nikolov, 2016]:

> Given a directed graph (digraph) $G(V,E)$ with a set of vertices V and a set of edges E, the Sugiyama algorithm solves the problem of finding a 2D hierarchical drawing of G subject to the following readability requirements:
>
> - (a) Vertices are drawn on horizontal lines without overlapping; each line represents a level in the hierarchy; all edges point downwards.
> - (b) Short-span edges (i.e., edges between adjacent levels) are drawn with straight lines.
> - (c) Long-span edges (i.e., edges between nonadjacent levels) are drawn as close to straight lines as possible.
> - (d) The number of edge crossings is the minimum.
> - (e) Connected vertices are placed as close to each other as possible.
> - (f) The layout of edges coming into (or going out of) a vertex is balanced, i.e., edges are evenly spaced around a common target (or source) vertex.

Having these requirements, the Sugiyama algorithm solves the problem by separating it into 4 distinct steps [Kobourov, 2012]:

- **Step 1:** Preparatory step for transforming the input digraph G into a proper hierarchy.

  - **Step 1.1:** Transform the input digraph G into a directed acyclic graph (DAG) by reversing the direction of some edges.
  - **Step 1.2:** Transform the dag into a multilevel digraph, called a hierarchy, by partitioning V into l levels (or layers) $V1, V2, \ldots, V_l$ such that for each edge $e = (v, w) \in E$ if $v \in V_i$ then $w \in V_{i+1}$. Levels are drawn on horizontal lines which determine the y-coordinates of the vertices.
  - **Step 1.3:** Transform the hierarchy into a *proper hierarchy* by introducing dummy vertices along long-span edges; one dummy vertex at each crossing of a long-span edge with a level.
- **Step 2:** For each level V_i, specify a linear order σ_i of the vertices in V_i with the goal of minimizing the total number of edge crossings.
- **Step 3:** Determine the x-coordinates of the vertices subject to requirements (c), (e), and (f) while preserving the linear order in the levels.
- **Step 4:** Draw G in a 2D drawing area where dummy vertices are removed and the long-span edges are restored.

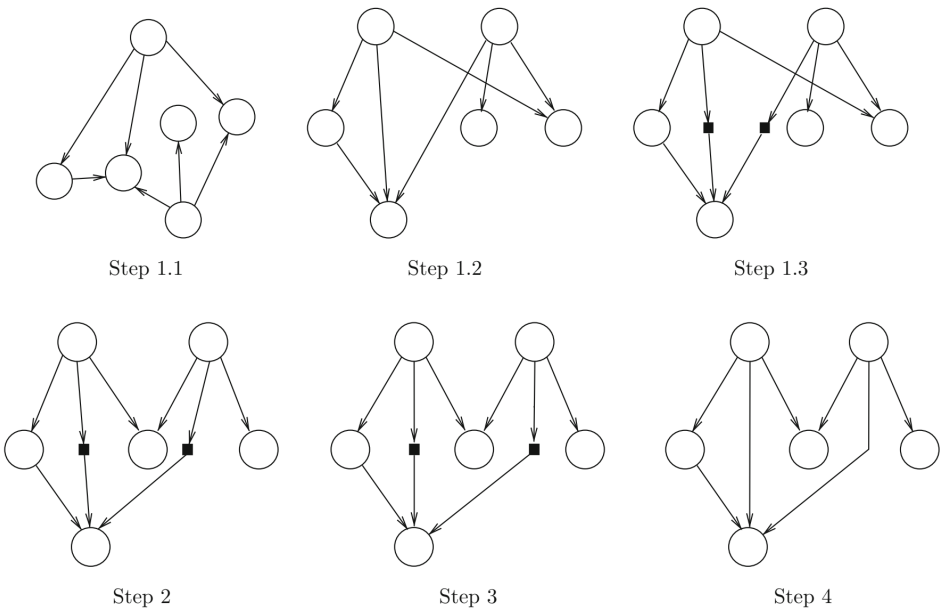


Figure 3.1: Application of the Sugiyama framework steps [Nikolov, 2016]

Sugiyama et al.'s [Sugiyama et al., 1981] key work focuses on providing efficient heuristics for steps 2 and 3, which are hard to solve given the readability constraints. Figure 3.1 shows an example of the application of the framework's steps.

Each of the steps in the algorithm defines a problem that needs to be solved, the output of one step defining the input of another [Kobourov, 2012]. As mentioned above, some of them are difficult to solve, i.e. steps 2 and 3, and others can be quite easy if the only requirements are the ones listed above, i.e. steps 1.1 and 1.2. However, those steps can become NP-hard once more

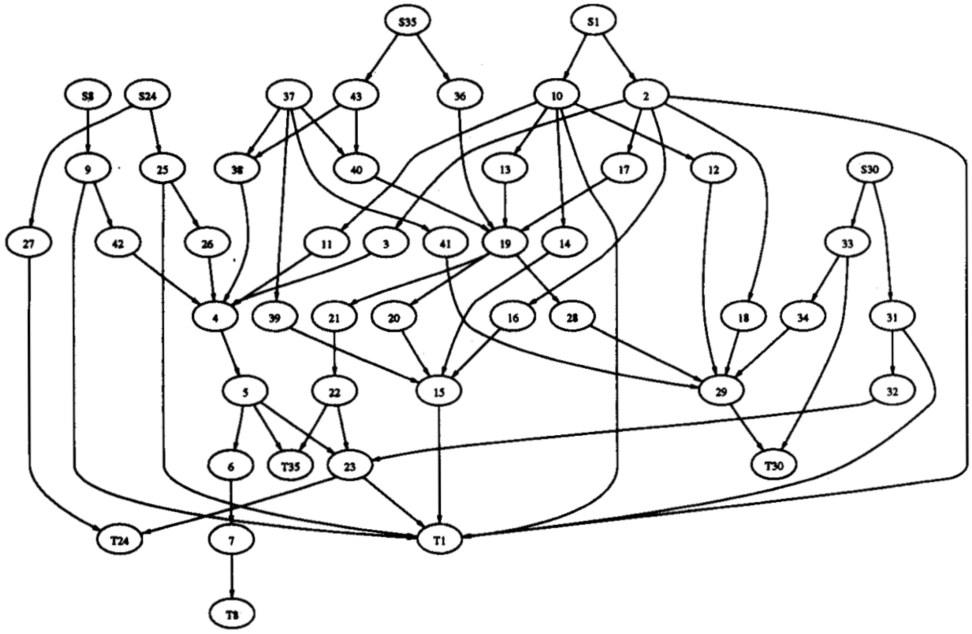


Figure 3.2: Example of *dot* output [Gansner et al., 1993]

constraints are applied (e.g. in step 1.1, minimizing the number of reversed edges is NP-hard). Other steps are trivial to execute, i.e. steps 1.3 and 4. Since each step is a different problem, there have been many studies over the years looking into each specific problem, defining improved heuristics for solving them more efficiently, or to fit them to a certain context better, this latter point will be explored in further sections below. Because of this "modular" nature of the Sugiyama algorithm, it is usually called the "Sugiyama Framework" [Kobourov, 2012].

Gansner et al. [Gansner et al., 1993] also describe an algorithm to layout directed graphs. It uses a four-pass approach similar to the Sugiyama algorithm. The first pass ranks each node with discrete ranks; the second pass orders the nodes within each rank to minimize edge crossings; the third pass assigns coordinates to each node, and the fourth pass calculates the control points for the path of each edge. The differences with the Sugiyama algorithm lie in the heuristics used to order the edges within layers as well as the method to calculate the x-coordinates of nodes. Another difference is that long-span edges are rendered as splines (whose control points are defined in the fourth pass), while the Sugiyama algorithm simply draws them as a sequence of straight lines. This described algorithm has been used on *dot* [Gansner et al., 2015], a graph drawing program. Figure 3.2 shows an example output for the *dot* program.

3.1.1.3 Radial Layouts

Radial layouts draw trees in a way that expands radially and outward (see Fig 3.3). These layouts usually focus on a root node, displaying the nodes related to it around it. The distance of each node to the center is usually related to the level of the node in the tree, although some implementations of radial layouts use other metrics to define that distance (e.g. see Section 3.1.2.2).

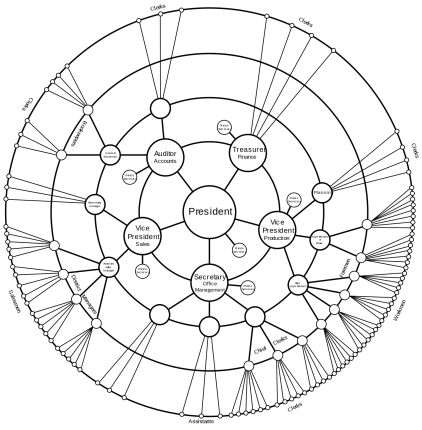


Figure 3.3: Example of a radial layout [Smith, 1925].

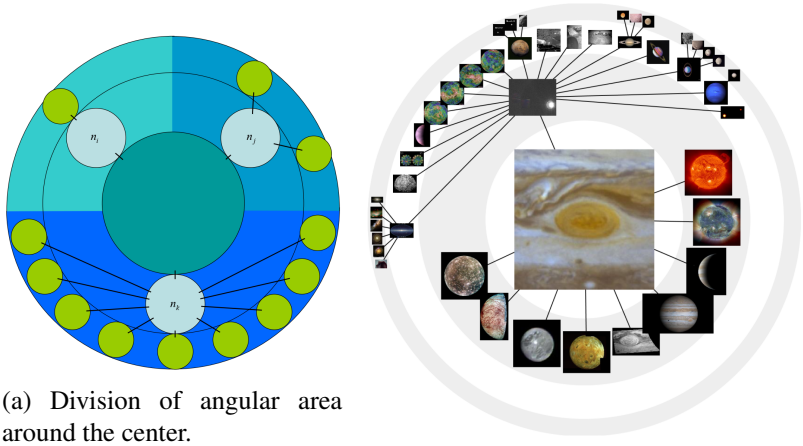
An example of the application of radial layouts is the work by Jankun-Kelly and Ma, who look into visualization techniques for graphs with visual nodes [Jankun-Kelly and Ma, 2003]. They contribute with MoireGraphs, a visualization technique that combines a radial layout with navigation techniques to aid people in navigating graphs with visual information. The layout creates a spanning tree originating from a root node, using a breadth-first search, and separates the circle around the root into rings, called levels. Each level corresponds to the distance from the node to the root in terms of hops through the spanning tree. The width of each level decreases the further away it is from the center, i.e. the root node, meaning the nodes appear smaller the further they are from the root. The angular area around the center is divided to give enough space for the subtrees of each child of the root. Figure 3.4a shows an example of this happening. Results can be seen in Fig 3.4b, which shows a MoireGraph of images from the NASA Planetary Photojournal.

3.1.2 Genealogical Graph Layout

Displaying genealogical information through graphs adds some extra things to look out for when laying these out. There are certain kinds of information that can be intuitively displayed through the intelligent layout of genealogical graphs. Some of these are the chronological aspects of the information as well as the generations of the individuals. These characteristics are some of the things users expect to easily notice in these visualizations.

3.1.2.1 Dual Trees

McGuffin and Balakrishnan [McGuffin and Balakrishnan, 2005] present a visualization of genealogical graphs with the goal of it being easy to interpret and scaling well. For this, the authors look into subsets of data that are familiar to users and how they can be displayed or combined in a manner that is easy to interpret and scales well. The achieved solution was a so-called dual tree, which is a merge of a tree of ancestors and a tree of descendants of different individuals. It works by obtaining a tree of ancestors $A(x)$ and a tree of descendants $D(y)$, such that y is one of x's ancestors, and merging them and displaying them in an indented tree layout (see Fig. 3.5).



(a) Division of angular area around the center.

(b) Layout example.

Figure 3.4: Moire graphs visualizations [Jankun-Kelly and Ma, 2003].

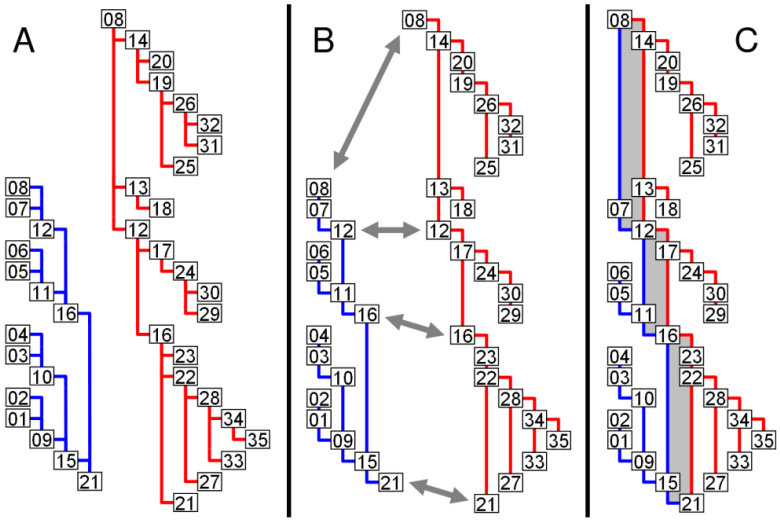


Figure 3.5: Creation of a dual tree, through merging an ancestor tree (in blue) and a descendant tree (in red) [McGuffin and Balakrishnan, 2005].

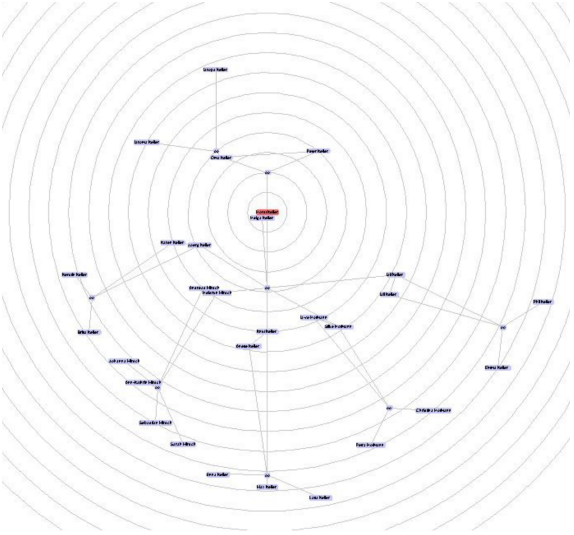


Figure 3.6: Example of the radial and force directed layout [Keller et al., 2010].

Evaluation of the dual tree visualization was done through a single informal session with a practicing genealogist, and it involved free-form exploration by the user, demonstration by the author, and the completion of certain tasks. From this session, the user reported the visualization being clear and easy, although the depiction of relationships was unfamiliar and took getting used to. There was also a comment pointing out the lack of a connection between spouses, which other conventional diagrams show.

3.1.2.2 Radial and Force Directed Approach

Keller et al. [Keller et al., 2010] developed a family tree representation combining two different layout techniques: radial and force-directed layouts. The representation displays a sub-graph centered around a single individual, showing their ancestors and descendants. The algorithm for the representation runs in two phases. Firstly a radial layout of the graph is computed with the individual at the center, with some constraints:

- ancestors are placed on the top and descendants are places on the bottom half of the diagram.
- the distance to the center is used to represent time, so, the older an ancestor and the younger a descendant are than the individual, the further way from the center they are placed.

Secondly, a force-directed layout algorithm is used to reduce the clutter of the nodes and keep the diagram more aesthetically pleasing. Figure 3.6 shows an example output of the approach.

This study provides some insight into how different approaches can be merged together to create new visualizations. The strengths of different approaches can be used to tackle different stages of a layout algorithm. This approach uses a radial layout to set the initial structure and a force-directed layout to improve its aesthetics.

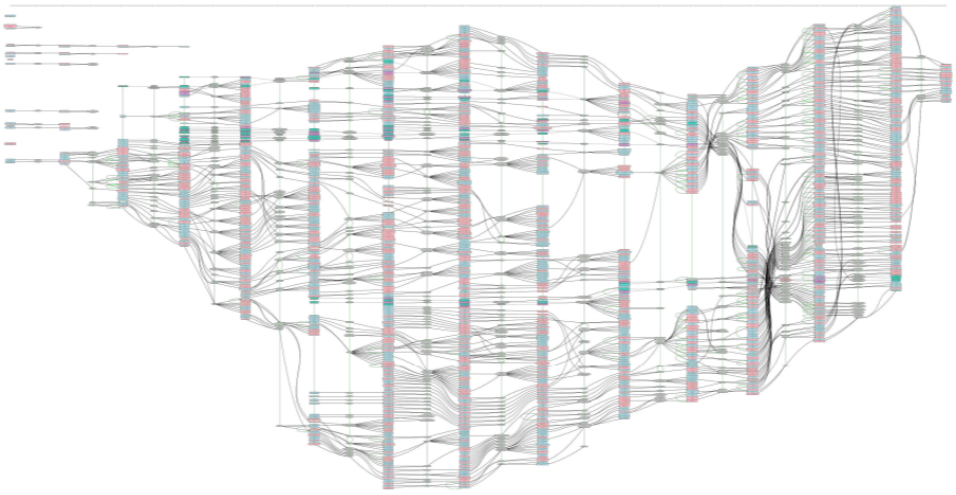


Figure 3.7: Example layout of Mařík's approach, applied on a family tree consisting of 1671 people[Mařík, 2016]

3.1.2.3 Hierarchical Layout Approach

As mentioned previously, genealogical information can be mapped to a directed acyclic graph. This opens the door for hierarchical layout methods to be taken advantage of.

Mařík did exactly this,[Mařík, 2016] picking up on the algorithm by Gansner et al.[Gansner et al., 1993], and defining an algorithm to assign for node rank assignment, and an algorithm for ordering the nodes within ranks, taking the characteristics of a genealogical graph into consideration. As per Mařík's words [Mařík, 2016]:

> A genealogical graph is an acyclic bipartite directed graph $G(V_P, V_M, E)$ with two sorts of nodes, people V_P and marriages/partnerships V_M. The edges E are directed from parent nodes to marriage nodes and from marriage nodes to children nodes.

The node rank assignment works to preserve generation separation, i.e. siblings sit in the same rank, and ancestors sit in ranks above their descendants. The in-rank node ordering keeps siblings clustered while parents can be mixed. This leads to easier identification of families, as the clustering of siblings makes the edges coming from marriage nodes more parallel. Figure 3.7 shows the result of the application of the new algorithm on a large family tree.

The drawback of this work is that Mařík did not write the algorithm from scratch, choosing instead to use the developed algorithms to generate constrained input for the *dot* program [Gansner et al., 2015] that influences it to generate the desired layout. This is not a perfect approach as the program might choose a different ordering if it finds it improves its target metrics, which might go into conflict with metrics relevant for genealogical graphs.

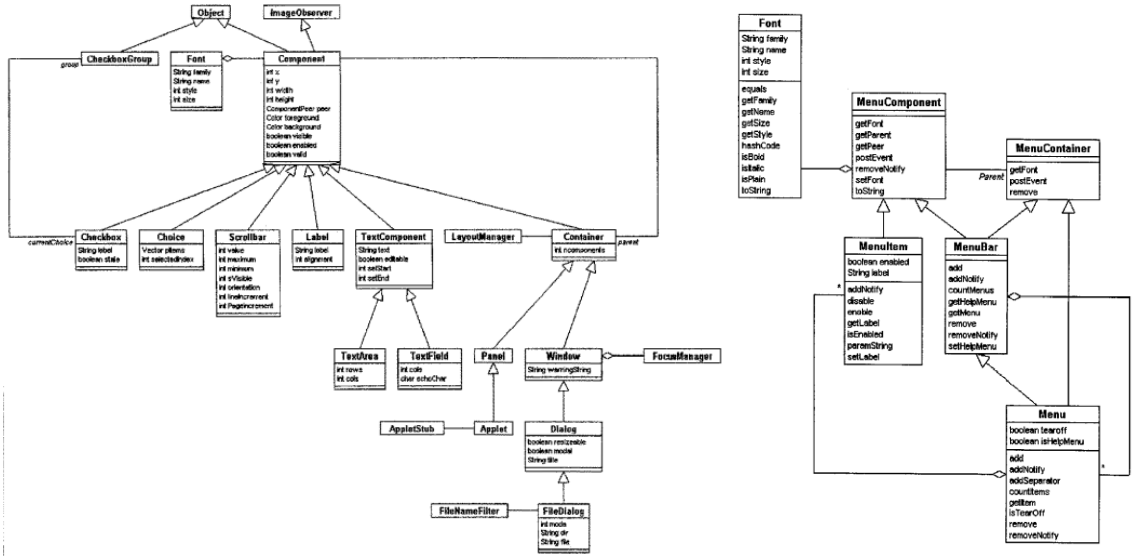


Figure 3.8: Examples of results from Seeman's algorithm. [Seemann, 1997]

3.1.3 UML Class Diagram Layout

There are many studies into the layout of UML class diagrams, with many different approaches used.

3.1.3.1 Hierarchical Approach

The hierarchical approach is based on the use of algorithms like the Sugiyama algorithm [Sugiyama et al., 1981] to layout a subset of the diagrams map. This approach bases itself on the fact that inheritance relations are usually drawn in an upwards direction. So, hierarchical layout algorithms are used to ensure this constraint.

An early example is Seemann's work [Seemann, 1997], where he developed an approach to drawing UML class diagrams by making use of hierarchical layout algorithms. Their approach is based on the condition that sub-classes must be drawn under their respective super-classes to emphasize the inheritance relationship between them. Their developed algorithm first computes a sub-graph containing all classes involved in inheritance relationships and lays it out with the Sugiyama algorithm, obtaining a hierarchical layout of these classes that satisfies the aforementioned condition. In the next steps, all remaining nodes are organized into sets of nodes related to each of the sub-graph nodes and incrementally added to the layout. The last steps adjust the node positions and create orthogonal paths for the association edges. Figure 3.8 shows examples of the resultant layout.

Eichelberger [Eichelberger, 2006] also presents an algorithm for laying out UML class diagrams based on hierarchical algorithms, along with an edge crossing reduction strategy tailored to UML class diagrams. His approach relies on the same constraint that inheritance relations should be emphasized but also takes into account other aspects like edge crossing, edge length and compactness. Figure 3.9 shows an example layout produced by Eichelberger's algorithm.

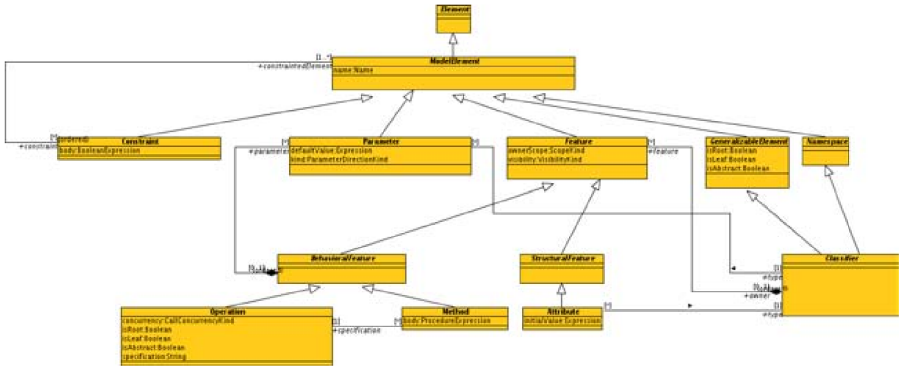


Figure 3.9: Example layout of Eichelberger's algorithm [Eichelberger, 2006].

3.1.3.2 Rank-Directed Approach

Hu et al. [Hu et al., 2012] describe a UML class diagram layout method called the rank-directed method. With this method, the authors' aim is to create a layout that emphasizes classes of higher importance. Here they assume that classes with more relations are more important. Their method consists of ranking all classes in the graph and separating it into sub-graphs, each of these sub-graphs containing one of the most important classes along with their more closely related classes. Each of these sub-graphs is then laid out using a magnetic force-directed algorithm that keeps the most important class as a central node. The final layout is computed by creating a new graph where its nodes are the created sub-graphs, and edges are created to represent the edges that connect nodes in different sub-graphs. This graph is then laid out using a force-direct approach as well. Figure 3.10 shows a comparison between a hierarchical method and the rank-directed method. The authors mention that through this layout, there is a sacrifice of aesthetics for better semantic readability.

3.1.3.3 Topology-Shape-Metrics Approach

Eiglspeger et al. [Eiglsperger et al., 2003] introduce an algorithm that uses a different approach to hierarchical and rank-directed layouts. It uses the topology-shape-metrics approach. The algorithm they specify aims to create orthogonal drawings of UML class diagrams (Fig. 3.11). Orthogonal graph drawings are graph drawings in which edges are drawn as a combination of horizontal and vertical segments. This approach, like the Sugiyama hierarchical approach, defines several steps:

- **Planarization** — Determines the topology of the graph by creating a planar embedding of it. The process of planarization involves converting a non-planar graph into a planar one by replacing edge crossings with dummy nodes.
- **Orthogonalization** — Determines the overall shape of the graph by determining the shapes of its edges along with the angles they do between each other at their endpoints. Since the resulting drawing is orthogonal, these angles are increments of 90°.

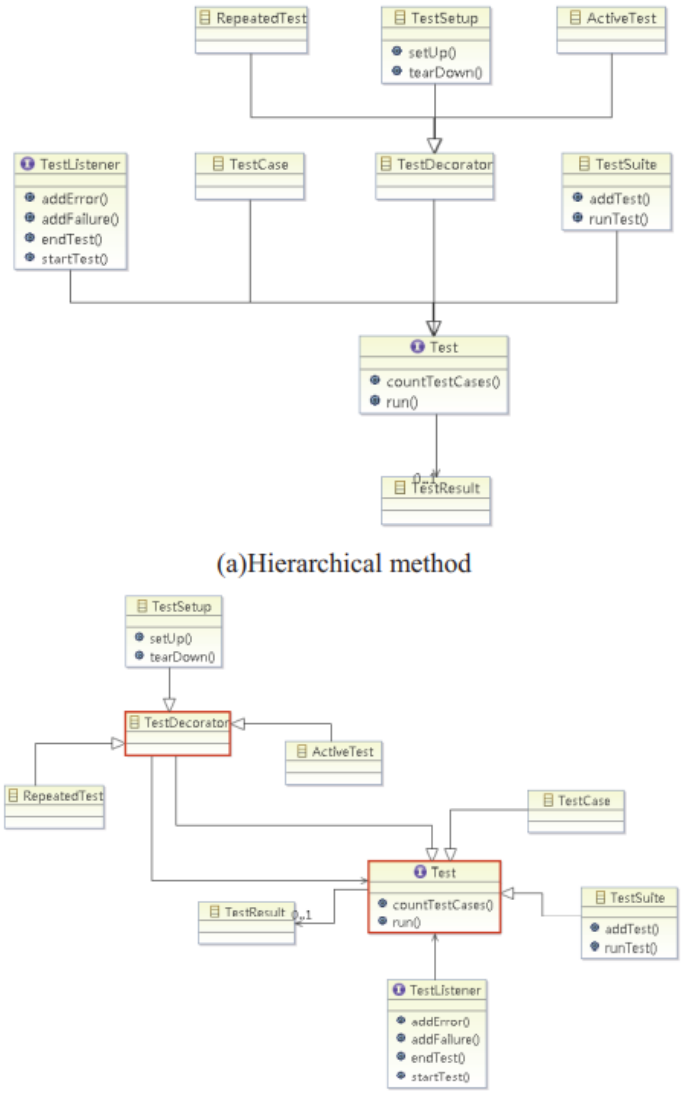


Figure 3.10: Comparison between a hierarchical method and the rank-directed method [Hu et al., 2012]

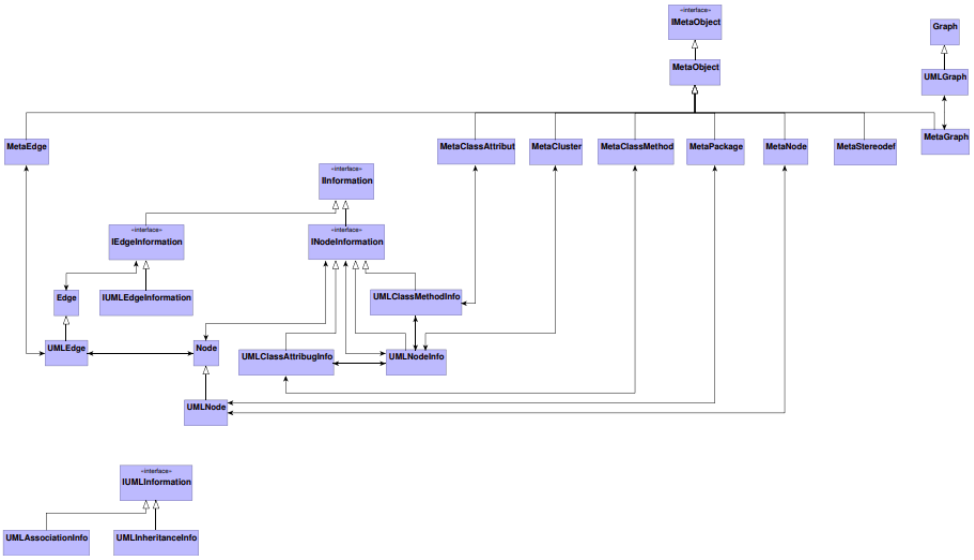


Figure 3.11: Example layout of the Topology-Shape-Metrics approach [Eiglsperger et al., 2003]

- **Compaction** — Final coordinates are assigned to nodes and edge bends.

The approach used in this algorithm models classes as nodes and the relations between them as edges. It allows for the definition of certain edges in the graph to be directed, meaning the graph is a mixed graph. The algorithm will try to draw these directed edges in the upwards direction, meaning that the planar embedding that is calculated by the first step is a mixed upward planar embedding.

Each of the steps in the algorithm is a complex problem in itself. Eiglspeger et al. provide, in their work, references to solve each one. The following sections will be used to cover each step and the algorithm used to solve the associated problems.

Before running the algorithm, some pre-processing is done on the graph. Firstly, the connected components of the graph are calculated. The algorithm actually runs on each of these components, so any following steps assume the graph is connected. Secondly, the directed edges are to see if they induce an acyclic subgraph. If they don't, edges are marked as undirected until they do. The directed edges are also checked to see if the subgraph they induce is connected. If not, some undirected edges are marked as directed by the use of a minimum spanning tree algorithm.

3.1.3.4 Planarization

The planarization step creates a mixed upward planar embedding of the graph. This is necessary to ensure that the graph can be drawn on the plane without edge crossings. If the graph is not planar, then it is planarized, i.e. converted into a graph that is planar by replacing its edge crossings with dummy nodes.

The method used to obtain a mixed upward planar embedding of the UML diagram's graph follows an incremental planarization method developed by Eiglsperger and Kaufmann [Eiglsperger and Kaufmann, 2001]. This method follows two main steps:

1. Finds a sub-graph of the original graph which is planar. A graph is planar if it admits a planar embedding. This sub-graph must include all nodes from the original graph. Edges from the original graph that are not in the sub-graph are leftover.

2. Incrementally add the leftover edges back into the sub-graph. Any edge crossings that result from adding a new edge are resolved by splitting the edges into sub-segments, having a auxiliary *crossing node* representing the crossing.

The method to find the maximal planar sub-graph follows a modified version of an algorithm by Goldschmidt and Takvorian (GT) [Eiglsperger and Kaufmann, 2001] [Goldschmidt and Takvorian, 1994]. The algorithm goes as follows:

1. Compute an ordering of the nodes of the graph, and arrange them in a line according to the ordering (Fig. 3.12b).

2. Obtain, for each edge, the positions of the nodes they connect. These positions define the range that the edge covers.

3. Create an auxiliary conflict graph, which contains one node for each edge in the graph, two nodes are connected if the corresponding edges cross. Considering $\pi(v)$ is the position of v in the ordering and two edges $e_1 = (a,b)$ and $e_2 = (c,d)$ such that $\pi(a) < \pi(b)$ and $\pi(c) < \pi(d)$. Edges e_1 and e_2 are crossing if $\pi(a) < \pi(c) < \pi(b) < \pi(d)$ or $\pi(c) < \pi(a) < \pi(d) < \pi(b)$. Figure 3.13a shows the generated graph from the ordering in Fig. 3.12b.

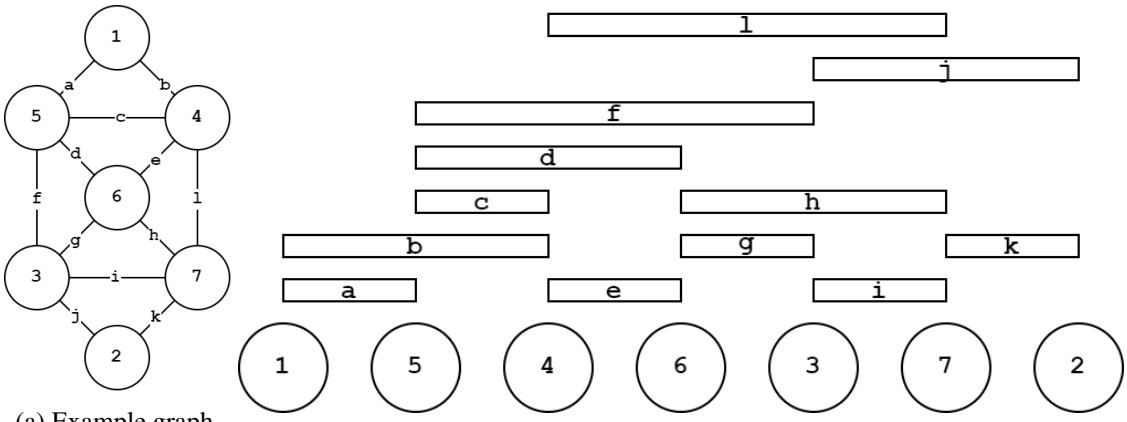
4. A two coloring algorithm is run on the conflict graph to obtain two distinct independent sets of edges. Each of these sets represent a side of the line, edges are routed through the side that the set represents (Fig. 3.13b). It is not guaranteed that all edges will be assigned a color, the ones to which this happens are considered as the input for the next step of the planarization.

The modification done the GT algorithm in the cited work introduces a new algorithm for Step 1 that ensures a mixed upward planar embedding is possible. The main point of the modification is ensuring that directed edges point all in the increasing order. This ensures that a mixed upward planar sub-graph because, if we arrange the nodes in a vertical line according to the ordering, the directed edges that are selected for the independent sets will be pointing up.

After calculating a mixed upward planar sub-graph, the process that re-adds the leftover edges functions over two phases. The first phase re-introduces the directed edges, and the second phase re-introduces the undirected edges. The method for adding directed edges to the graph is a constrained version of the method to add undirected edges so, for explanation sake, these steps will be covered in reverse order.

Undirected edge addition works by routing the *to be added* edges through the graph's faces with the help of an auxiliary routing graph. The algorithm works as follows (Fig. 3.14):

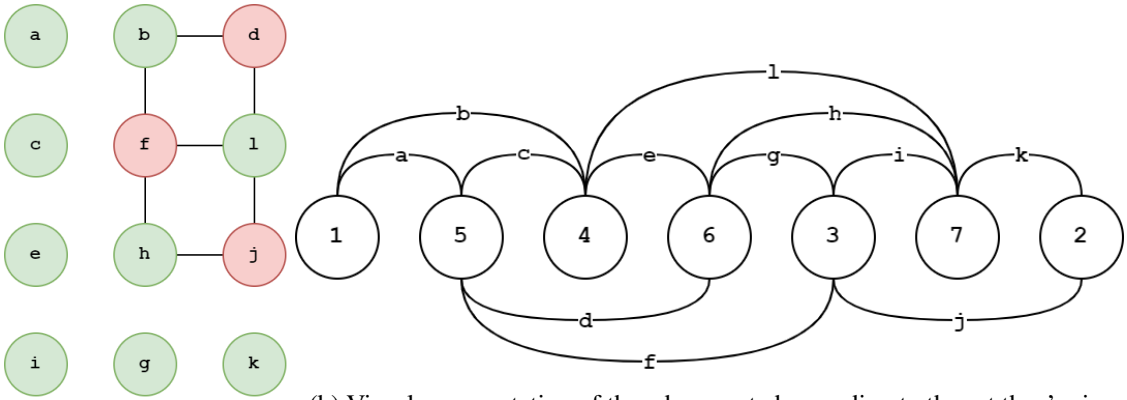
1. For each edge $e = (v,w)$ to add:



(a) Example graph.

(b) Graph's node arranged according to ordering, with an overlay of its edges.

Figure 3.12: An example graph, along with an ordering of it's nodes and edge ranges.



(a) Generated, two colored conflict graph.

(b) Visual representation of the edges routed according to the set they're in.

Figure 3.13: Two colored conflict graph and visual representation of the two independent sets.

(a) Initiate a routing graph.

(b) Create a node in the routing graph for each face in the graph.

(c) For each edge in the graph, create and edge with length 1 in the routing graph connecting the nodes that represent the faces the edge is adjacent to (Fig. 3.14b).

(d) Create auxiliary nodes v_v and v_w in the routing graph representing nodes v and w respectively, and connect them to the faces each node is adjacent to with edges of length 0 (Fig. 3.14c).

(e) Run a shortest path algorithm from v_v to v_w to obtain a path (Fig. 3.14d).

(f) All edges with length 1 in the obtained path are crossings that must be introduced into the graph.

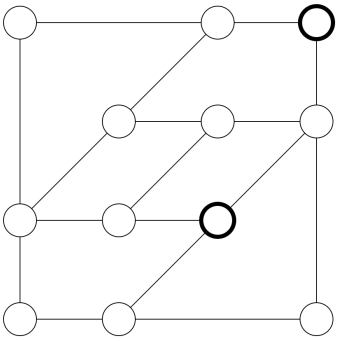
This algorithm must be run per leftover edge because with each new addition the planar embedding is changed. Certain nodes acquire new neighbours, while others have their neighbours change. This requires a recalculation of the graph's faces after each edge addition.

Directed edge addition further constrains the undirected edge addition. While undirected edges can be added independent of each other, directed edges can't, as introduction of crossing nodes introduces changes to the node ordering. This might result in cycles if edges are added later. Figure 3.15 shows an example of such a case, after the $(5,9)$ edge was added, edge $(3,4)$ can't be added without causing a directed cycle.

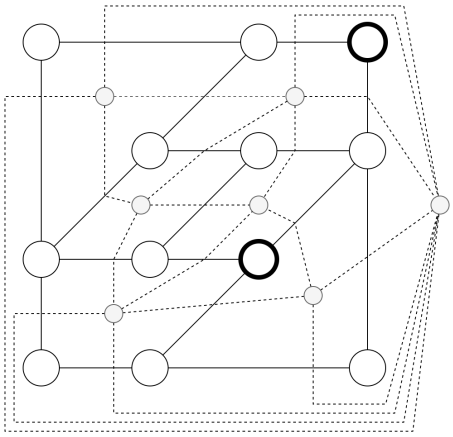
The algorithm for directed edge addition, works over directed s-t graphs, which are graphs with a single source and sink. There are methods to augment an upwards planar graph into an s-t graph, which will be explained in a later section.

Assuming the graph is an s-t graph. The algorithm avoids cycles by creating a layering of the graph. A valid layering l assigns layers to each nodes of the graph such that $l(w) > l(v)$ for every edge (v,w). For this layering the leftover edges are temporarily added into the graph. The routing graph is constructed as such:

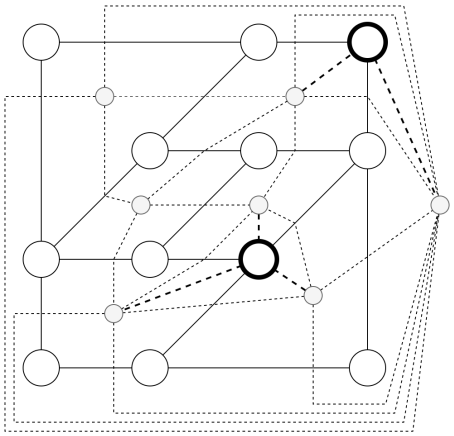
- The routing graph contains a node for each layer that each face spans.
- Two nodes in the same face, representing adjacent layers are connected by an edge of length 0 from the lower to the upper layer.
- Two nodes at the same layer, representing adjacent faces, are connected with an edge of length 1 if there is an edge in the s-t graph adjacent to both faces that spans that layer.
- The v_v and v_w of edge (v,w) that are added to the routing graph are connected to adjacent faces as such: v_v is connected to the "faces nodes" in the same layer as v with edges that go from v_v to the face nodes, and v_w is connected the face nodes in the layer directly below it with edges that go from the face nodes to v_w. All edges added in this manner have length 0.



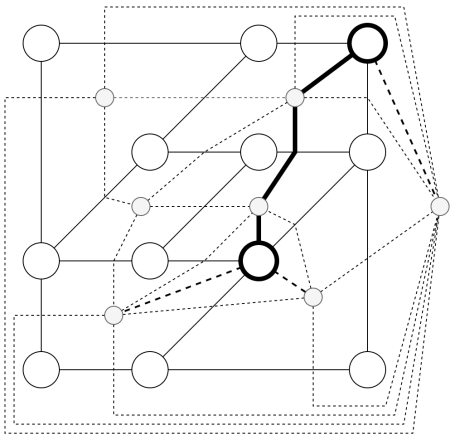
(a) Initial graph.



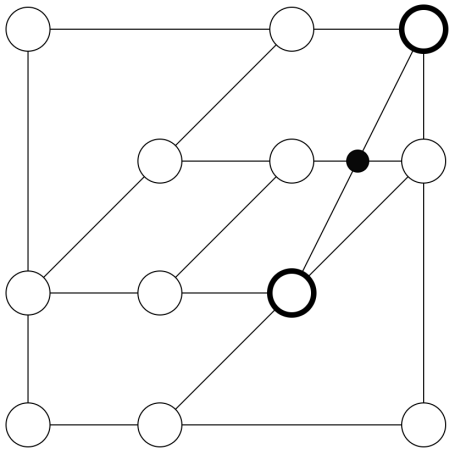
(b) Routing graph with connected face nodes.



(c) Connect nodes with adjacent faces.



(d) Applied shortest path algorithm.



(e) Edge added with crossing node in black.

Figure 3.14: Visualization of the undirected edge addition algorithm adding an edge between the highlighted nodes.

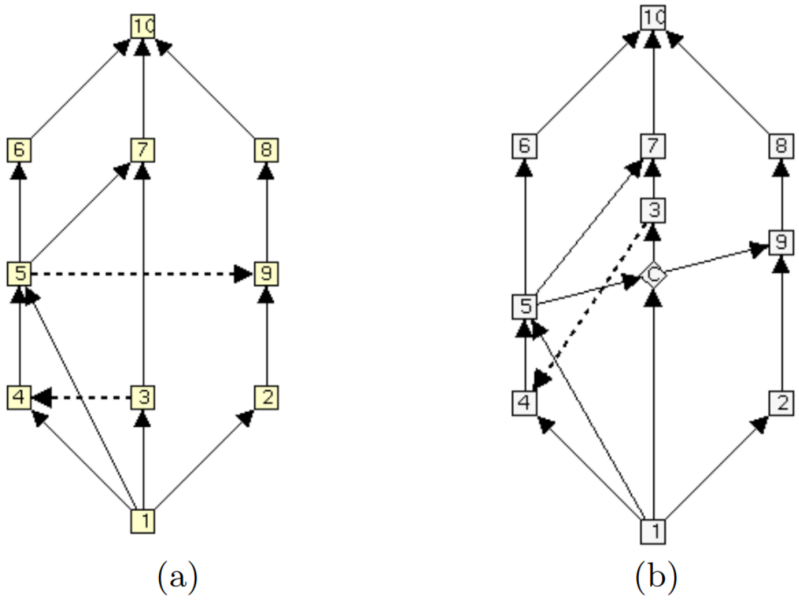


Figure 3.15: Example of directed edge addition causing a cycle [Eiglsperger and Kaufmann, 2001].

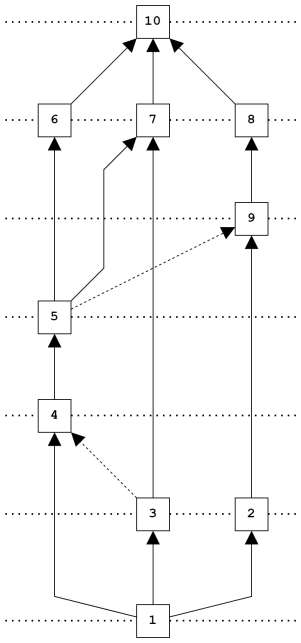
The outer face of the graph is also split into two separate faces, the left side and the right side. The resulting routing graph doesn't contain edges in decreasing layer order, which implies that the routing graph is directed. And, once again, edges of length 1 represent a crossing.

The graph over which the algorithm runs will most likely be a result of an augmentation process that converts it into an s-t graph. This augmentation process temporarily adds edges into the graph. Since crossing over these edges doesn't mean crossing over edges of the original graph, every edge that represents a crossing over an edge added during the augmentation process has a length of 0.

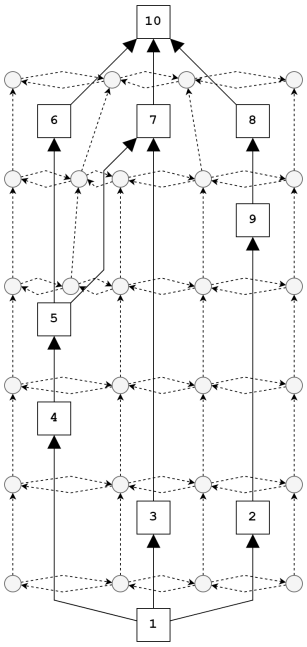
After constructing the routing graph, the algorithm can proceed in the same way as the undirected edge insertion one. Using a shortest path algorithm to determine the route of the edge to be added. Just like the undirected edge addition, this algorithm is also run for every edge that needs to be added for the same reasons. Figure 3.16 visualizes a run of the algorithm for edge $(5, 9)$ from the example shown in Figure 3.15.

Finally The overall algorithm for the planarization of the UML class diagram graph is as follows:

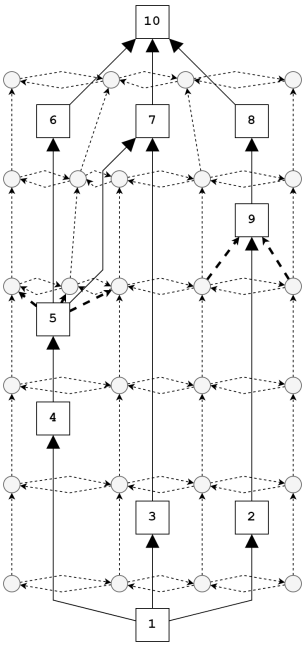
1. Calculate a mixed upward planar sub-graph.
2. Make undirected edges in the sub-graph temporarily directed according to the ordering in the sub-graph calculation.
3. Augment sub-graph into an s-t graph.
4. Run directed edge insertion algorithm for each directed edge that isn't in the sub-graph.



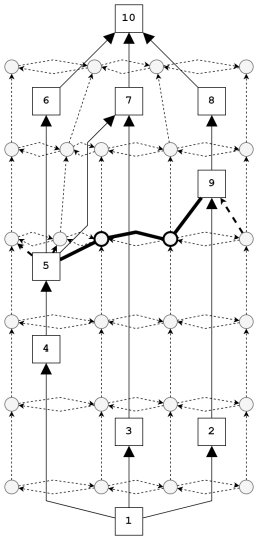
(a) Graph layering.



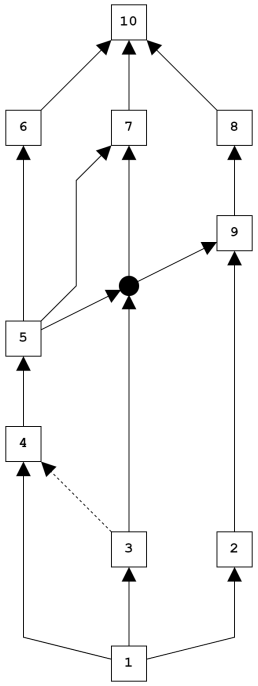
(b) Routing graph construction.



(c) Connecting nodes to face nodes.

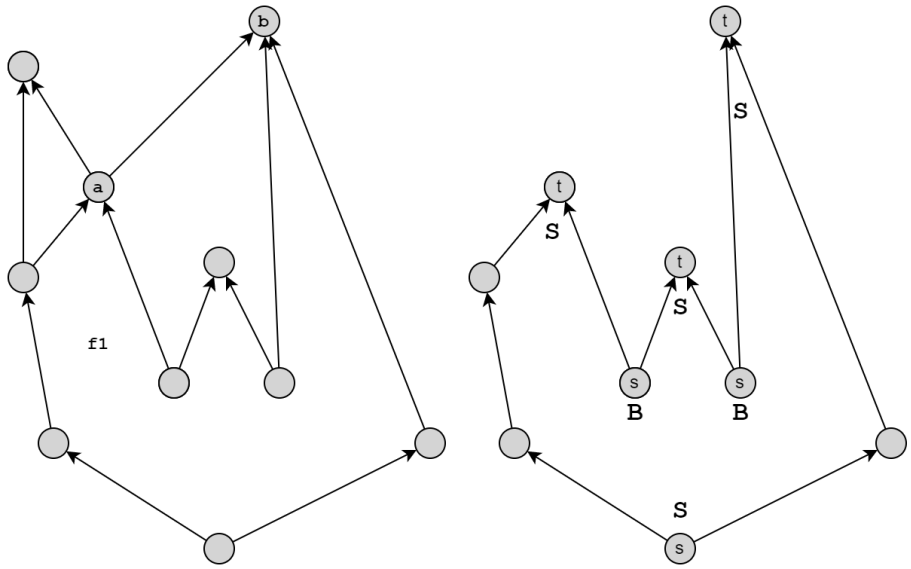


(d) Apply shortest path algorithm.



(e) Add edge, resolve crossings with crossing nodes.

Figure 3.16: Visualization of the directed edge addition algorithm adding edge $(5,9)$ with extra leftover edge $(3,4)$. Face nodes are offset from their layers for visual clarity.



(a) Embedding. (b) $f1$ switch assignment.

Figure 3.17: An upwards planar embedded graph and the assignment of the switches to face $f1$.

5. Remove edges inserted in augmentation process.

6. Run undirected edge insertion algorithm for each undirected edge that isn't in the sub-graph.

3.1.3.5 Augmenting Graph into S-T Graph

As mentioned above, directed edge addition only works over s-t graphs. In order to generalize the algorithm, the graphs need to be augmented into an s-t graph.

Bertolazzi et al. in a work to create a method for testing the upward planarity of tri-connected graphs, present a method of augmenting a bi-connected graph into an s-t graph [Bertolazzi et al., 1994]. The augmentation method requires that the graph is upwards planar.

The first step of the method is to identify the switches of each face. Switches are nodes that are either sources or sinks, i.e. nodes without *in* edges or *out* edges, respectively. Graph switches are switches on the whole graph's scope. Face switches are switches only on the face scope. As an example, node b in Figure 3.17a is a graph switch since it only has *in* edges, node a is a face switch in face $f1$ since it only has *in* edges in the face.

After determining all face switches, it is required to determine the kind of angle they form inside their faces. These angles are distinguished between big (B) angles and small (S) angles. Figure 3.17b exemplifies the assignment of the angles to the switches face $f1$.

The next step of the method creates, for each face f, a cyclic sequence of symbols, "obtained by traversing f in clockwise order and assigning s_B and t_B (B-symbols) to source-switches and sink-switches labeled B in f, and s_S and t_S (S-symbols) to source-switches and sink-switches labeled S in f." [Bertolazzi et al., 1994]. After this the following algorithm is performed:

1. If the face has exactly one source and one sink, stop.

2. Find a sequence (x, y, z) of one B-symbol followed by two S-symbols, once found:
   (a) If $(x, y, z) = (s_B, t_S, s_S)$, add edge (v_z, v_x).
   (b) If $(x, y, z) = (t_B, s_S, t_S)$, add edge (v_x, v_z).
   (c) Remove x and y from the cyclic symbol list.
3. Repeat from Step 1.

This algorithm must be run on all internal faces. Once it is done, all of the graphs sources and sinks will be present on the external face. The sources will be present on the bottom side, and the sinks will be present on the top side. It is also expected that all of them will form big angles in the external face. From here, a source and a sink node can be created, to which the all sources and sinks will be connected, respectively. It is worth pointing out that the source and sink nodes only need to be created if the graph has more than one of the respective kind. See Figure 3.18 for a step by step visualization.

3.1.3.6 Orthogonalization

The orthogonalization step defines the more concrete shape requirements of the graph. It takes as input a planar embedding of the graph and outputs a quasi-orthogonal representation of the graph.

In his work, Tamassia [Tamassia, 1987], describes a 1:1 correspondence between the orthogonal representation of a graph G and a flow over some network N_G. The network contains the set of nodes $U = \{s\} \cup \{t\} \cup \{U_v\} \cup \{U_f\}$, where s and t are the source and sink of the network, U_v contains a node for every vertex of G and U_f contains a node for every face of G. Paraphrasing from Fößmeier and Kaufmann, the arcs between the network's nodes are the following [Fößmeier and Kaufmann, 1996]:

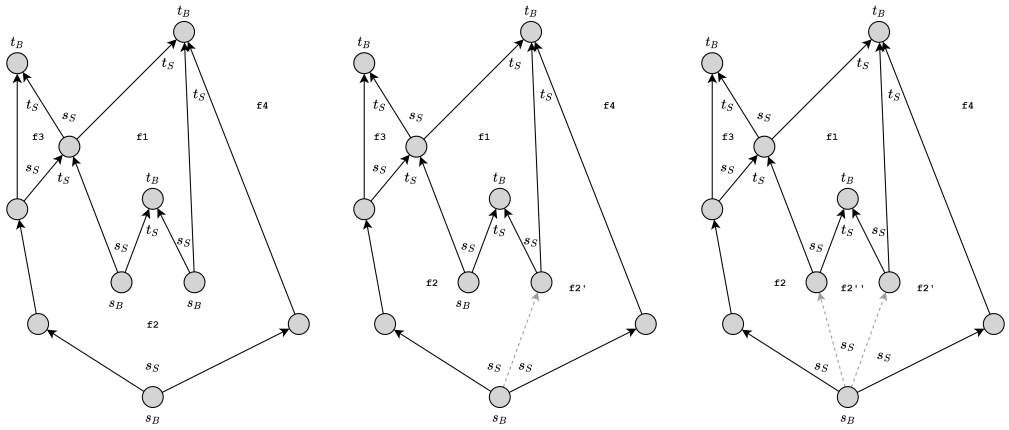
a) arcs from s to nodes $v \in U_v$ with cost 0 and capacity $4 - deg(v)$, for every node with degree below or equal to 4.

b) arcs from s to nodes $f \in U_f$, where f represents an internal face of G with $deg(f) <= 3$; these arcs have cost 0 and capacity $4 - deg(f)$; $deg(f)$ for a face f denotes the number of edges in its clockwise list of edges, including the repeated ones;

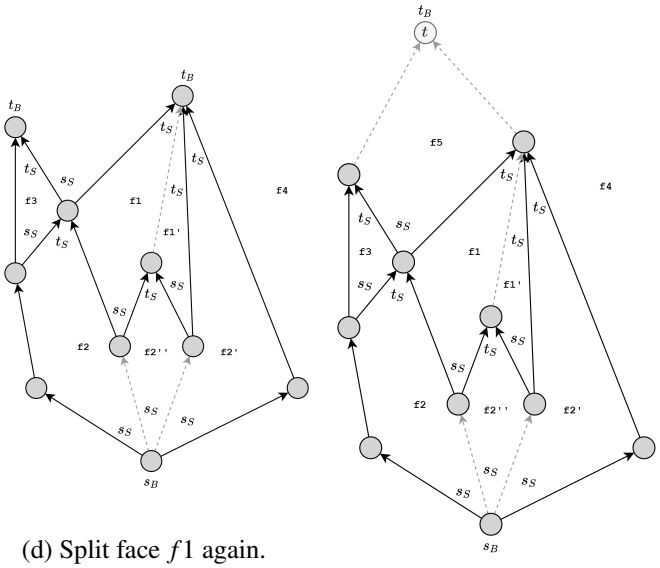
c) arcs from nodes $f \in U_f$ representing the external face or representing internal faces f with $deg(f) >= 5$ to t; these arcs have cost 0 and capacity $deg(f) - 4$ if f is an internal face and capacity $deg(f) + 4$ for the external face;

d) arcs of cost 0 and capacity ∞ from nodes $v \in U_v$ to nodes $f \in U_f$, if v is incident to an edge of the face f;

e) arcs of cost 1 and capacity ∞ from a node $f \in U_f$ to a node $g \in U_f$, whenever the faces f and g of G have at least one common edge.



(a) Symbol assignment. (b) Split face $f2$. (c) Split face $f2$ again.



(d) Split face $f1$ again.

(e) Add auxiliary sink.

Figure 3.18: Step by step s-t graph augmentation.

Each unit of flow in the network represents an angle of 90°. "The flow on the arcs in d) defines the angles of the orthogonal representation: If $x_{v,f}$ is the flow from the node $v \in U_v$ to the node $f \in U_f$ then the angle at vertex v in face f is $(x_{v,f}+1)\times 90°$" [Fößmeier and Kaufmann, 1996]. For arcs in e), the flow $x_{f,g}$ from node $f \in U_f$ to node $g \in U_f$ represents the number of 90° angles that edges shared by faces f and g make inside face f. Considering $b(v,w)$ to represent the capacity of arc (v,w). Any flow of value $\sum_u b(s,u) = \sum_w b(w,t)$ through the network with cost B, corresponds to an orthogonal representation with exactly B bends. Which means that applying a minimum cost flow algorithm over the network will yield a representation with the minimum amount of bends.

To obtain a quasi-orthogonal representation of the graph, the network needs to be extended to support 0° angles between edges. Fößmeier and Kaufmann, developed such an extension [Fößmeier and Kaufmann, 1996]. Since a flow of 0 from a node to a face represents an angle of 90°, 0° angles are represented by a flow of -1. This is represented as a flow of 1 in the opposite direction, from the face to the node. Some additional arcs are added to the network:

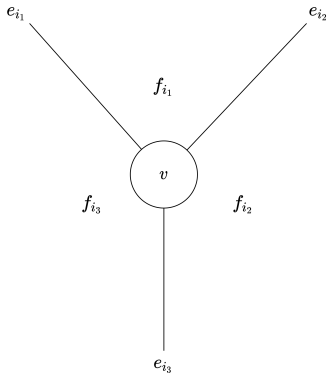
f) arcs of cost 0 and capacity $deg(v)-4$ from nodes $v \in U_v$ to t, for all nodes in U_v with $deg(v) > 5$.

g) arcs of cost 0 and capacity 1 from a node $f \in U_f$ to a node $v \in U_v$, whenever there is an arc of type d) from v to f.

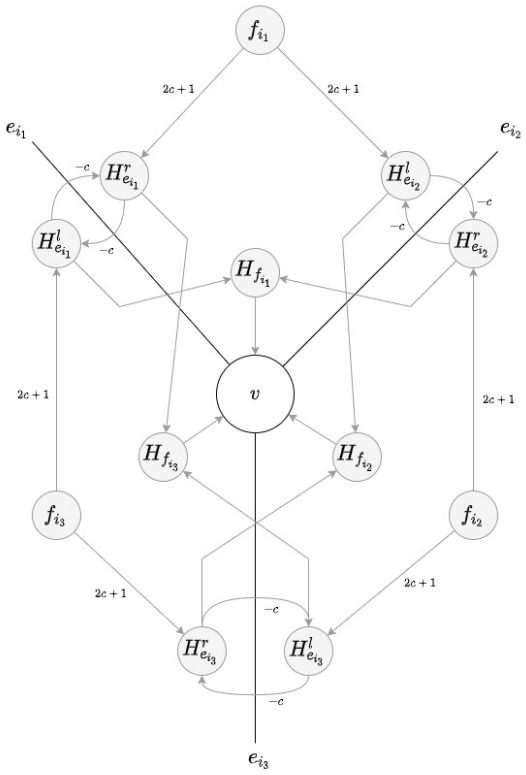
Arcs of type g) aren't actually directly created in the network as there are some restrictions around when flow can go through them, this relates to how the graph needs to be correctly drawn. The overall graph layout algorithm follows the Kandinsky model, in which the *bend-or-end* property states that at most one edge attached to each side of a vertex can be straight, these edges are called the middle edges. This means that when there is a flow from a node $f \in U_f$ to a node $v \in U_v$, there must also be a flow from a node $g \in U_f$ to f, this ensure that at most one edge is straight, to ensure this condition arcs of type g) can be replace by arcs of type h) which go from g directly to v. Another cautionary step to take is that all edges to the left of the middle edges must bend to the left, and all edges to the right of the middle edge must bend to the right. These restrictions impose special prohibited cases of flow on the network. These cases are avoided by creating a special construction that replaces arcs of type h).

The construction that establishes arcs of type h) and ensures the compliance of the necessary restrictions goes as quoted [Fößmeier and Kaufmann, 1996]:

> Let v be a node in U_v and $f_{i_1},\ldots,f_{i_k}$ an ordered list of the faces around the vertex v in the graph (e.g. in clockwise order); let $e_{i_1},\ldots,e_{i_k}$ be the edges that separate these faces such that e_{i_j} separates face $f_{i_{j-1 \bmod k}}$ and face f_{i_j} (e.g. Fig 3.19a). (...)
>
> Then we add for every edge e_i, being incident to v two nodes $H^l_{e_{i_j}}$ and $H^r_{e_{i_j}}$, where $H^l_{e_{i_j}}$ ($H^r_{e_{i_j}}$) corresponds to the arc of type h) crossing edge e_{i_j} in clockwise (counter-clockwise) order around v; further $H_{f_{i_j}}$ are new nodes in the network for every face f_{i_j}.
>
> New arcs are:



(a) Faces and edges in clockwise order [Fößmeier and Kaufmann, 1996].



(b) Auxiliary construction.

Figure 3.19: Auxialiary construction for 0° angles. Network angles and edges in gray. All capacities are 1, costs that are not indicated are 0.

- Arcs with capacity 1 and cost $2c+1$ (c having a suitable value) from f_{i_j} to $H^r_{e_{i_j}}$ and to $H^l_{e_{i_{j+1 \bmod k}}}$, i.e. to the edges of the graph corresponding to the arcs of type h) starting in face f_i; and crossing a bounding edge of this face.
- Arcs with capacity 1 and cost 0 from the nodes $H_{f_{i_j}}$ to node v; the arcs of these two types replace the arcs of type h).
- Arcs with capacity 1 and cost 0 from node $H^l_{e_{i_j}}$ to node $H_{f_{i_j}}$ and from node $H^r_{e_{i_{j+1 \bmod k}}}$ to node $H_{f_{i_j}}$, i.e. from two auxiliary nodes for two neighboured edges to the auxiliary node for the face between them. (...)
- Arcs with capacity 1 and cost $-c$ from node $H^l_{e_{i_j}}$ to node $H^r_{e_{i_j}}$ and vice versa. Every pair of such arcs defines a cycle of cost $-2c$ and thus a cost minimum path from a node f_{i_j} to a node v has cost $2c+1-2c=1$ corresponding to one necessary bend as in the case at the arcs of type h). Every time a second flow unit passes one of the nodes $H^l_{e_{i_j}}$ or $H^r_{e_{i_j}}$, the arcs with negative cost are already satisfied and thus the path from f_{i_j} to v has cost 2c + 1.

The only worry here is making sure the constant c is an appropriate value to ensure the construction avoids violating the drawing restrictions, having a value of $c > B$ works well enough.

Figure 3.19 shows a visualization of the construction.

This concludes the basic functionality of an orthogonalization step. However, Eiglsperger et. al. add some additional steps. The shapes of the directed edges are pre-calculated through a linear complexity algorithm, this will ensure the that they are drawn in the upward direction. Since some edges' shapes area already pre-defined, an extension to the orthogonalization step is used to enforce these shapes. This extension, developed by Brandes et. al., adds additional devices to the flow network N_G that provide hints of which angles nodes should make inside a face and which bends edges should have. This can be considered as providing a graph sketch for the algorithm to work with [Brandes et al., 2002].

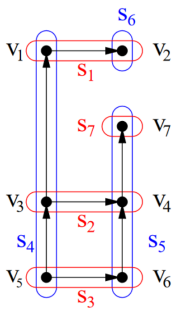
3.1.3.7 Compaction\Coordinate Assignment

The compaction phase is responsible for assigning edge coordinates to nodes and edge bends. The compaction algorithm used for this step in Eiglsperger et. al.'s work, is specified in a paper by Eiglsperger and Kaufmann [Eiglsperger and Kaufmann, 2002]. Their paper tackles the compaction problem for graphs types of graphs with increasing complexity. First, a solution for compaction of 4-graphs with point nodes is presented. Secondly, the solution is expanded to support n-graphs ($n > 4$), i.e. support for 0° angles. Lastly, the algorithm is further expanded to support variable sized nodes, this is the most important part of the algorithm since nodes that represent classes in UML class diagrams are all rectangular and have different sizes.

Before starting the coordinate assignment process, some prepossessing must be done. The coordinate assignment algorithm takes a quasi-orthogonal representation of the graph and for any tuple (e_r, a_r, b_r) in the orthogonal representation where b_r is not empty, the corresponding edge e_r is split into segments connected by dummy nodes representing the edge bends. This is necessary as the algorithm relies on all edges in the graph being straight. The rest of the explanation assumes that this splitting has been performed.

Using the information in the orthogonal representation H, it is possible to determine the orientation of all of the edges in the graph. The orientation dir of an edge e can be represented as $dir(e) = u$ if the edge is vertical and $dir(e) = r$ if the edge is horizontal. For each edge $e = (v, w)$, there are two entries in H, in one entry the edge is traversed from v to w and in the other it is traversed in the reverse direction from w to v. Using this information it is possible to calculate in which absolute direction $d \in \{up, down, left, right\}$ an edge is traversed during face traversal. This information can be directly appended to the tuples in H. For the rest of the section, the graph's edges are assumed to all be pointing to the *right* if they are horizontal and *upwards* if they are vertical, this is to simplify some explanations.

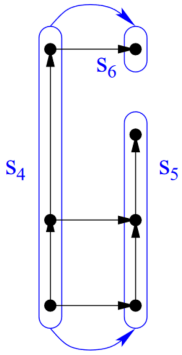
With the orientations of every edge in the graph calculated, two sub-graphs are created $G_r = (V, E_r)$, containing only the horizontal edges of the graph, and $G_u = (V, E_u)$, containing only the vertical edges in the graph. Denoting S_r the set of connected components in G_r, and S_u the set of connected components in G_u, the elements of theses sets can be called, horizontal segments or vertical segments, respectively. Two segments are adjacent if they share a node. $seg(e)$ is the



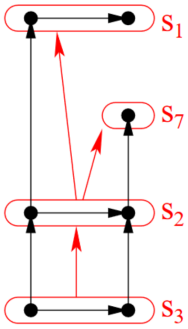
(a) Graph.

Segment	Nodes	Edges
s_1	$\{v_1, v_2\}$	$\{(v_1, v_2)\}$
s_2	$\{v_3, v_4\}$	$\{(v_3, v_4)\}$
s_3	$\{v_5, v_6\}$	$\{(v_5, v_6)\}$
s_4	$\{v_1, v_3, v_5\}$	$\{(v_1, v_3), (v_3, v_5)\}$
s_5	$\{v_4, v_6, v_7\}$	$\{(v_4, v_6), (v_4, v_7)\}$
s_6	$\{v_2\}$	$\emptyset$
s_7	$\{v_7\}$	$\emptyset$

(b) Segment definition.



(c) D_r



(d) D_u

Figure 3.20: Example of graph's vertical and horizontal segments, with respective constraint graphs [Eiglsperger and Kaufmann, 2002].

segment to which edge e belongs, $vert(v)$ is the vertical segment to which node v belongs and $hor(v)$ is the horizontal segment to which v belongs.

Then, a pair of constraint graphs that define the ordering of the segments is created. The constraint graphs $D_u = \{S_r, A_u\}$ and $D_r = \{S_u, A_r\}$, have their edges defined as follows:

$$A_u = \{(hor(v), hor(w)) : (v,w) \in E_u\}$$
$$A_r = \{(vert(v), vert(w)) : (v,w) \in E_r\}$$

A shape description $S = (D_r, D_u)$ combines both constraint graphs to define the overall shape requirements of the graph. If the shape description is complete, a standard layering algorithm is able to obtain a valid orthogonal drawing of the graph. By creating a layering of each of the constraint graphs, integer coordinates can be assigned to each segment. A node v's x coordinate is determined by the layer of $vert(v)$ and the y coordinate by the layer of $hor(v)$.

As mentioned above, it is only possible to obtain a valid drawing if the shape description S is complete. Since each node is only contained in one horizontal and one vertical segment, this means that two adjacent segments only geometrically overlap each other on the node that they share. A complete shape description ensures this by making sure non adjacent segments are properly separated. If a shape description is not complete, the application of the layering of the segments will cause node overlap in the final drawing (e.g. Fig. 3.21). To resolve this problem, a super-set of S that is complete, denominated the complete extension of S, can be computed using heuristics.

The method to extend the shape description S relies on decomposing the faces of the graph into rectangles by looking for patterns of bends by iterating through the orthogonal representations of each face. By denoting angles of 90° with a '0' and angles of 270° with a '1', a list l of tuples $t = (a, s, d)$ can be created, with $a \in \{0, 1\}$ representing the bends, s being the segment of the edge that forms the angle and d being the direction in which the edge was being traversed. Nodes that

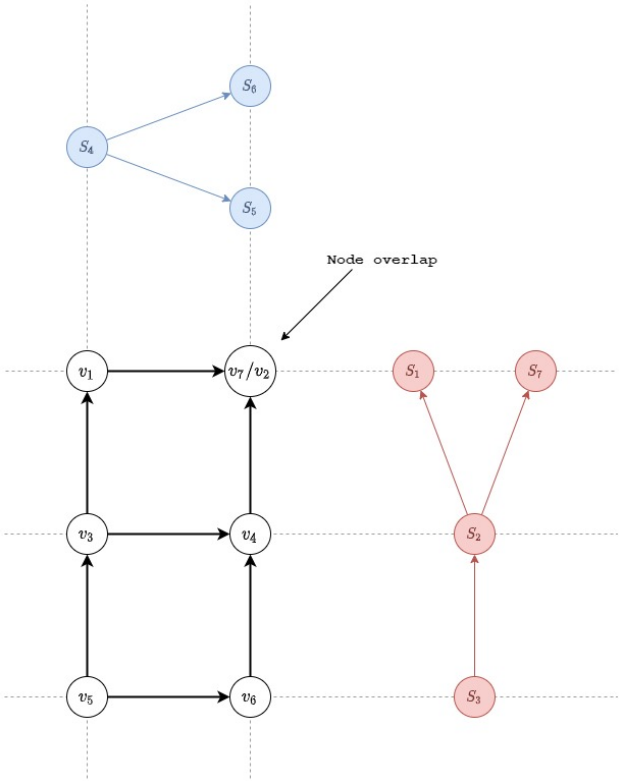


Figure 3.21: Example of an incomplete shape description producing an invalid drawing. The layering of the constraint graphs is correct, however, nodes v_2 and v_7 overlap. Constraint graphs from Fig. 3.20.

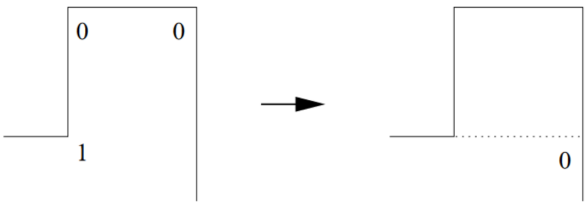


Figure 3.22: Cutting a rectangle from a face [Eiglsperger and Kaufmann, 2002].

form a 360° angle inside of a face belong to a segment that only contains them (e.g. Fig 3.20a S_6 only contains v_2 and S_7 only contains v_7). In this sense, angles of 360° are converted into two consecutive tuples with $a = 1$, the first one contains the segment that makes the 360° angle, and the second contains the aforementioned segment. 180° angles are not considered bends. The list l for the external face of the graph in figure 3.20a would be:

$$\{(1,s_4,down),\ (1,s_3,right),\ (1,s_5,up),\ (1,s_7,left),\ (0,s_5,down),$$
$$(0,s_2,left),\ (0,s_4,up),\ (1,s_1,right),\ (1,s_6,up),\ (1,s_1,left)\}$$

The list l is repeatedly searched for patterns of '100', this pattern represents a rectangle that can be cut and replaced with a '0' angle (e.g. 3.22). This rectangular decomposition doesn't change the original graph G, but just adds extra edges to the shape description (e.g. Fig. 3.23).

After decomposing all faces of the graph, including the external face (Fig. 3.24), the shape description of the graph will be complete. A layering of the segments will now yield valid orthogonal drawings (e.g. Fig. 3.25).

To support graphs where nodes can have degree higher than 4, the concept of sub-segments is introduced. Since nodes in 4-graphs have at most one edge attached per side, a segment can be modeled as a path with a single orientation. In n-graphs this is not guaranteed since nodes can have multiple edges in one side. So segments become more of a collection of paths. As quoted [Eiglsperger and Kaufmann, 2002]:

> We call two edges (u,v) and (v,w) a right-join (left-join) if they have the same direction and between them in the cyclic order (reverse cyclic order) there are only edges with different directions.

Sub-segments are paths of edges with the same direction where every two consecutive edges are right-join or every two consecutive edges are left-join. Sub-segments can also be composed of a single edge. A sub-segment is maximal if it is not contained inside other sub-segments. The algorithm only handles maximal sub-segments, so any mention of a sub-segment in later parts assumes that it is maximal. Figure 3.26 show a segment in an n-graph along with its sub-segments.

The rectangular decomposition in n-graphs needs to be modified. First, in the definition of the list l of tuples $t = (a,s,d)$, s is now the sub-segment to which the edge belongs according to

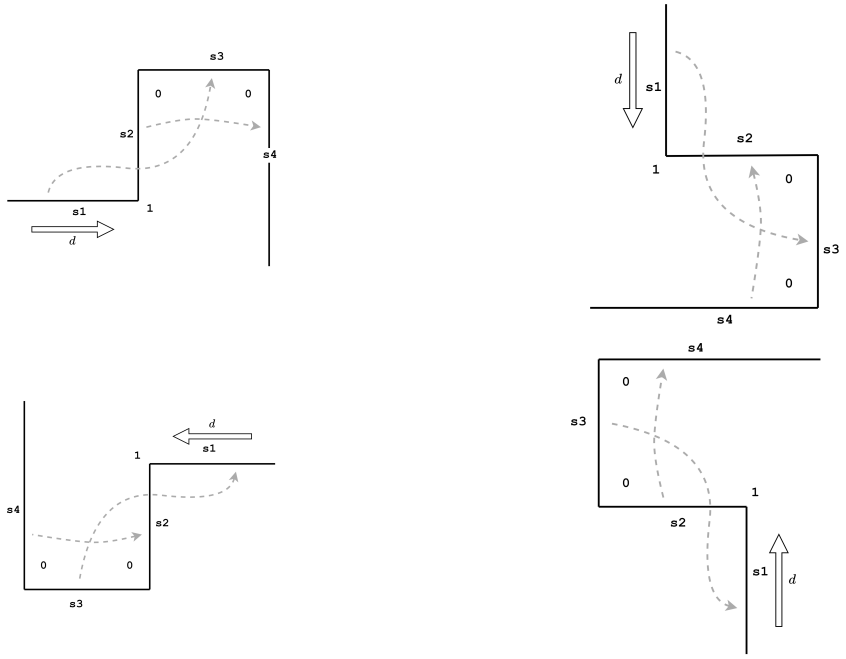


Figure 3.23: Example of the addition of new constraint edges during rectangle decomposition.

d. Since it needs to be able to handle 0° angles, those are denoted with a '-1'. Then, before performing regular rectangle decomposition, the '-1' bends need to be eliminated through a modified decomposition algorithm. This algorithm will look for the patterns '0-11', '1-10' and '1-11'.

Figure 3.27 shows how the different cases of zero angles are resolved. For the '1-11' sequence, there are two different ways it can be resolved, i.e. cases c) and d). Which one is chosen depends on some technical constraints.

After eliminating the zero angles with the above decomposition, the regular decomposition algorithm that was defined before can be run on the resultant list l to decompose the face into rectangles once more. The segments that are connected together are the segments corresponding to the sub-segments in the list's tuples. An exception is when a segment is connected with itself, in this case the connection in not made.

The adaptation of the algorithm to support nodes with different sizes involves the creation of a simplification of the original graph $G = (V, E)$. Denoting $width(v)$ as the width of a node v and $height(v)$ as the height. It is assumed that the of number edges attached to the top or bottom of a node v is lower than $width(v) + 1$ and the number of edges attached to the left or right is lower then $height(v) + 1$.

Considering the set $B \subset V$ of dummy nodes introduced at the start to represent bends in edges. All other nodes $V \setminus B$ are replaced with rectangular faces. For each edge e adjacent to v, a new node $p(e,v)$ is created which represents the port of e on v. Also, four corner nodes $nw(v)$, $ne(v)$, $sw(v)$ and $se(v)$ are created. Each node face has four adjacent node-segments: the top side segment $t(v)$, the bottom side segment $b(v)$, the left side segment $l(v)$, and the right side segment $r(v)$. The result of this process is graph $G_S = (V_S, E_S)$ with an orthogonal representation H_S. G_S is a 4-graph,

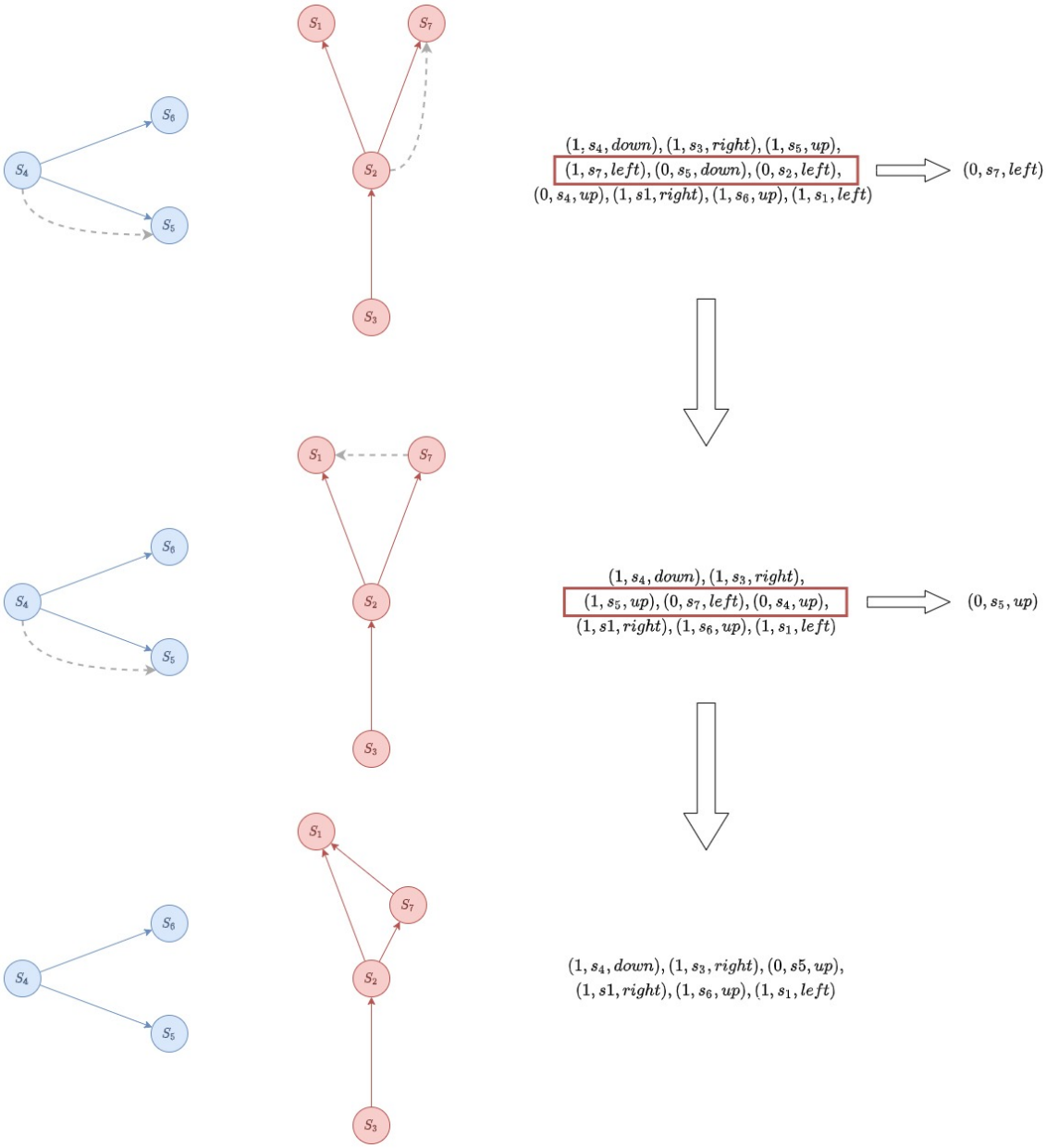


Figure 3.24: Rectangular decomposition applied to the external face of the graph in Fig. 3.20a. Edges added in each step in gray.

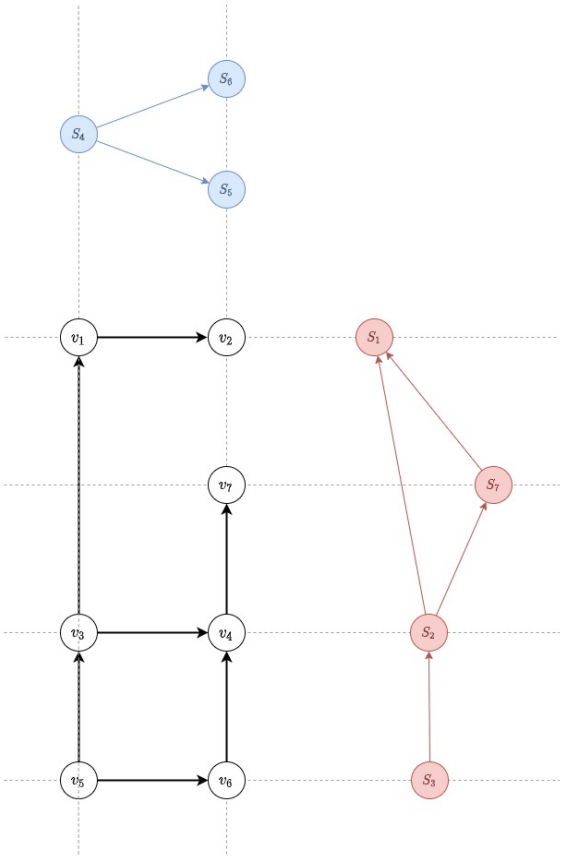


Figure 3.25: Layering of constraint graphs with a complete shape description now produce a valid drawing.

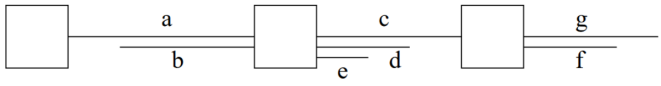


Figure 3.26: A segment in an n-graph. According to the definition, the sub segments are (a,c,g), (b,e), (d) and (c,f) [Eiglsperger and Kaufmann, 2002].

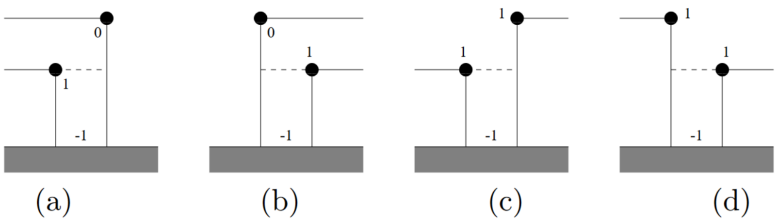


Figure 3.27: Rules to remove zero degree angles [Eiglsperger and Kaufmann, 2002].

however, a valid drawing of a 4-graph won't guarantee a valid drawing of G_S.

There are some constraints that must be imposed over creating a drawing of G_S, in order for it to map to a valid drawing of G:

1. The edges adjacent to corner nodes may have a length of 0.
2. The distance between the segments representing a node's sides must respect the node's dimensions.

For this, the constraint graphs in the shape description are refined by adding a length function. The shape description of G_S is defined as follows $S' = \{D'_u, D'_r\}$, D'_u is defined with $A'_u = A_u \cup N_u^+ \cup N_u^-$, with:

$$N_u^+ = \{(b(v), t(v)) : v \in V\}$$
$$N_u^- = \{(t(v), b(v)) : v \in V\}$$

D'_r is defined in the same way. The length function $length(e)$ is defined for A'_u as:

$$length(e) = \begin{cases} 0, & \text{if } e \in A_u, e \text{ is adjacent to a corner} \\ height(v), & \text{if } e \in N_u^+ \\ -height(v), & \text{if } e \in N_u^- \\ 1, & \text{otherwise} \end{cases}$$

The rectangular decomposition of G_S is made in tandem with the rectangular decomposition of G. Before doing that, some extra elements must be added to S', some definitions follow.

Considering $v \in V \setminus B$ and $d \in \{up, down, left, right\}$. The function $seg(v,d)$ is defined as such:

$$seg(v,d) = \begin{cases} r(v), & \text{if } d = up \\ l(v), & \text{if } d = down \\ t(v), & \text{if } d = left \\ b(v), & \text{if } d = right \end{cases}$$

Let $s = e_0, \ldots, e_r$ be a sub-segment, with $e_0 = (v_0, v_1), e_1 = (v_1, v_2), \ldots, e_r = (v_r, v_{r+1})$, $d \in \{up, down, left, right\}$, and $simple(e)$ a function that maps an edge e from G to the corresponding edge in G_s. The corresponding meta-segment $meta(s,d)$ is defined as:

$$\{seg(v_0,d), simple(e_0), seg(v_1,d), simple(e_1), \ldots, simple(v_r), seg(v_{r+1},d)\}$$

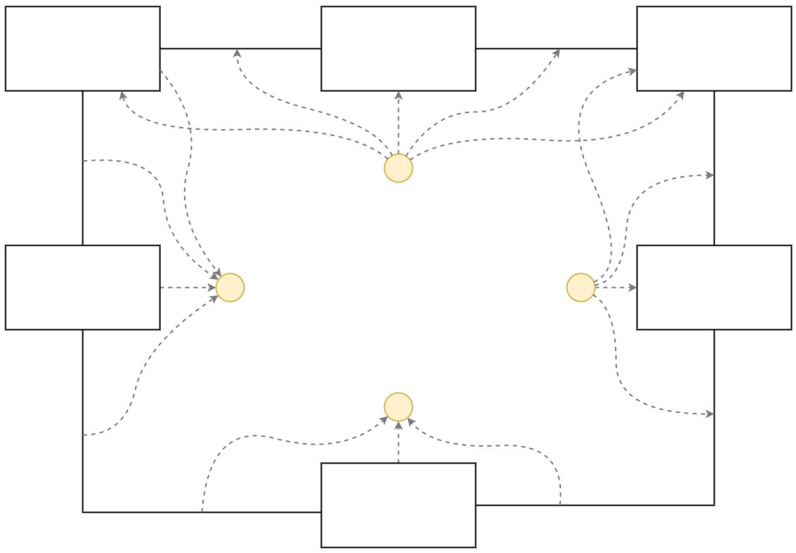


Figure 3.28: Example of face with $mn(s)$ nodes colored yellow. Edges connecting H_S segments to their corresponding meta segment nodes in dashed gray.

Now, for every face f of G, the zero angle elimination from above is run on f, this results in a list of tuples l, this list is generated from H, not H_S. For every tuple $t = (a,s,d)$, a meta segment node $mn(s)$ is created in the corresponding constraint graph, this node represents the boundaries of the meta segment inside the face. Edges are created connecting $mn(s)$ and every segment of the meta segment. These meta segment nodes are considered to be inside the face, so, the direction of the edges must defined in a way that they canonically point to the right or upwards (e.g. Fig 3.28). The length of these edges is 0. Lastly some additional edges are created between meta segment nodes and normal segments to properly separate nodes that might overlap at their corners (Fig. 3.29). Lastly, the rectangular decomposition is run on l, when rectangles are cut from faces in H, the corresponding meta segments in H_S are connected to enforce the same restrictions (e.g. Fig. 3.30).

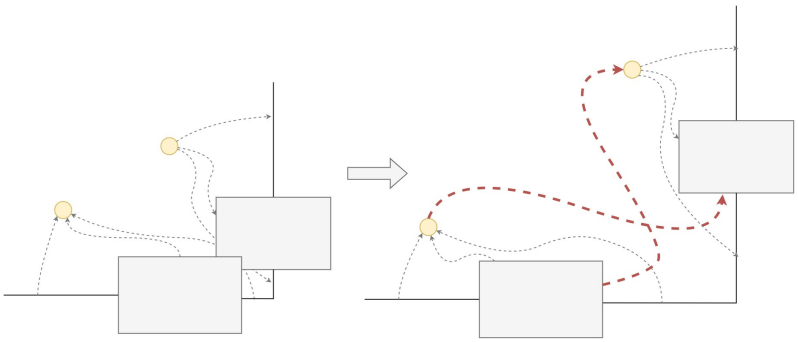


Figure 3.29: Example of two nodes that might overlap, and how it is avoided with the constraints. $mn(s)$ nodes in yellow, added edges in red.

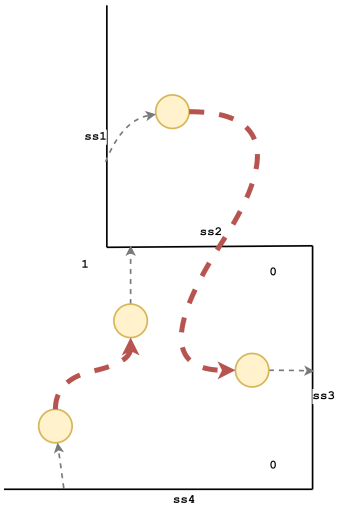


Figure 3.30: Example rectangle decomposition with meta segment nodes.

3.1.4 Summary

The algorithms for graph layout and respective resultant layouts depend on the objectives they are trying to achieve and the constraints they are trying to fit into.

Regarding Family Tree visualizations, users depending on them might have varying objectives. This reflects the varied approaches mentioned, which result in very different layouts. The Dual Trees approach's objective is to provide a clean and scalable way to browse through genealogical data while respecting some user necessities like preserving the generational ordering [McGuffin and Balakrishnan, 2005]. The radial+force-directed approach had a bigger emphasis on the temporal aspect, having the nodes' exact position be influenced by their timestamp data and making the tree visualization more pleasing [Keller et al., 2010]. These methods display only parts of the genealogical graph, while Mařík's work [Mařík, 2016] aims at a full view of the genealogical graph without hiding any information, which is beneficial in some cases. All approaches are good at displaying hierarchical information, which genealogical data can be categorized as. In comparison, the Dual Trees and radial+force-directed approaches are suitable for displaying clean layouts but lack the ability to display full graphs, while hierarchical graph drawing approaches like the one in [Mařík, 2016] can display the entire network of families but lack readability as the graphs start to scale up.

For UML class diagrams, the objectives behind their usage are very similar, i.e. understanding the static structure of a system. Even so, layout algorithms achieve different results due to differing philosophies of how they should be drawn. Hierarchical approaches strictly force inheritance relations to be drawn in an upwards fashion, making the layouts centered around these relations. This has the advantage of clean layouts with short edges when the diagrams rely a lot on inheritance. However, these approaches become inefficient when diagrams start to rely more on associations. Seemann even admits this in his work [Seemann, 1997]. The Topology-Shape-Metrics approach has a bigger concern with overall graph aesthetics, leaving the upwards drawings of inheritances

as a secondary concern. The layouts produced around these concerns end up being cleaner overall and with fewer bends, although edges can be on the longer side. The Rank-Directed approach by Hu et al. [Hu et al., 2012] has different objectives from the last two methods. While the last two methods were only concerned with clean layouts without awareness of the importance of each class, this rank-directed approach makes an effort to identify which parts of the system are closely related to ensure the classes belonging to those parts are drawn closer together, sacrificing some readability. In comparison, the Hierarchical and Topology-Shape-Metrics approaches give readable drawings of class diagrams, but their lack of understanding of the meaning behind the relations might be detrimental, while an approach like the rank-directed one helps users understand the most important parts of a system with a sacrifice in the cleanliness of the drawing.

3.2 Graph Interaction and Navigation

Some of the works presented above also feature interaction features to complement the produced visualizations. This section documents them.

3.2.1 Dual Tree Interactions

The dual tree layout work mentioned in Section 3.1.2.1 [McGuffin and Balakrishnan, 2005] also presents features for interacting with the visualization to allow for easy exploration and information extraction. The interaction features are centered around four basic operations: expanding and collapsing of a node's parents and expanding and collapsing of a node's descendants. The expansion of certain nodes will have side effects on the layout, e.g. expanding the parents of a node in the descendants tree will require the layout to be rearranged, nodes need to be moved, and some others will need to be hidden to keep the dual tree scheme. The expansion, collapse, and movement of the nodes are shown through 1-second animations. These animations have three stages: fade out nodes that need to be hidden, move nodes to new positions, and fade in nodes to be added. Although these animations may get complex, the authors believe a complicated animation is better than no animation at all and that users can benefit from tracking just a few nodes. Panning and zooming of the camera can be done manually or automatically through animations as well.

The selection of which nodes to expand or collapse is done through a "*subtree-drag-out* widget for "dragging out" subtrees to any depth" [McGuffin and Balakrishnan, 2005] (see Fig. 3.31). It works by using a secondary mouse button to click on a node, the node will be highlighted, and an initial widget will popup to select whether the user wants to work over the tree of ancestors (up) or the tree of descendants (down), the user then must drag in the direction of the tree they want to work on. After that, a second widget appears to select whether the users want to expand (down) or collapse (up) the tree. It allows the user to select the level to which expand or collapse while also displaying the level up to which the tree is expanded.

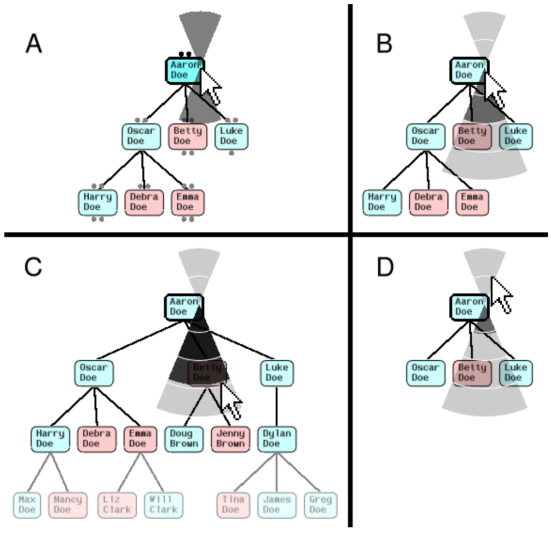


Figure 3.31: Dual Trees interaction widget [McGuffin and Balakrishnan, 2005].

3.2.2 Radial and Force Directed Interactions

The prototype for the work on the Radial+Force Directed visualization in Section 3.1.2.2 [Keller et al., 2010] also has some interaction features:

- **Dynamic Root Node Switching**: A user can change the root node of the visualization. By doing this, the focus point of the tree will change. When the root node changes, a transition between layouts will happen, certain parts of the tree will collapse, and others will expand to show the ancestors and descendants of the new root node. When changing the root to a child or parent of the root node, the transition will first hide sub-trees that need to be collapsed, transition the nodes to their new positions and then show the sub-trees that need to be expanded. The transition in the node positions is done by linearly interpolating the nodes' polar coordinates rather than their Cartesian ones. When the new root node is more separated from the current one, e.g. grandmother or grandson, the transition will play along the path between the nodes, e.g. when changing to the maternal grandmother, it will first transition to the mother node and then to the grandmother node.

- **Hover Tooltip**: Used to keep the visualization less cluttered by assigning a tooltip to each node. It provides additional information about each person, e.g. birthdates, and spouses.

- **Age Bracket Highligthing**: The visualization includes a slider that allows the user to highlight nodes within a certain age difference from the root node. The difference in age is calculated between a node and the root node, so positive values will highlight ancestors, and negative values will highlight descendants.

- **Search**: A text box that allows the user to search people by their names. Nodes are dynamically highlighted as the user types the search query. This feature allows the visualization to be much more scalable since nodes become easier to find.

- **Same Generation Highlight**: Similar to the age bracket highlighting. In this case, it highlights nodes within a certain generation in relation to the root node. It also uses a slider. Similar to the age bracket slider, positive numbers highlight ancestors, and negative numbers highlight descendants. One of the uses for this feature is the extraction of age trends within generations as well as the range of ages.
- **Cone Steepness**: Another slider is provided to change the steepness level of the tree. The steepness level dictates how spread out the nodes are. More spread-out nodes reduce clutter and make it easier to see each individual node. Having them less spread out allows for the fitting of more nodes in the visualization.
- **Zooming and Repositioning**: For better ease of use, other implemented features were zooming in and out of the tree using the right mouse click and the manual repositioning of the nodes, which the user can do by clicking and dragging the nodes, to better fit the user's preference. The repositioning is constrained so that the distance from the moved node to the root node stays the same and reflects the relative age. So, the repositioning can be seen more as a rotation of the node.

3.2.3 MoireGraphs Interactions

The work on the MoireGraphs visualization shown in Section 3.1.1.3 [Jankun-Kelly and Ma, 2003] also introduces several interactive features to allow a user to fine-tune the visualization:

- **Changing Focus Strength**: The focus strength dictates the size of the root node in relation to subsequent layers. Higher values will make the root node larger, with layers towards the periphery becoming smaller and losing detail, while smaller values will reduce the root node's size with periphery layers becoming bigger and gaining detail.
- **Radial Rotation**: The feature allows a user to offset the angle of the nodes' polar coordinates. This is useful when the edge direction has significance, e.g. temporal information.
- **Level Highlighting**: While hovering the mouse over a certain level, the level gets allocated twice its available space, resulting in its nodes increasing in size while other levels become smaller. There is also a visual queue that colors the respective level. Abrupt transitions are avoided by animating the changes in size from highlighted to non-highlighted states (and vice-versa). Useful for comparisons between certain levels and the root node.
- **Secondary Foci**: Secondary foci can be chosen to compare individual nodes with the root node. A node that is chosen as a secondary focus increases in size relative to its siblings. However, the layout doesn't change, which can result in the node occluding its siblings. Secondary foci are chosen by hovering the mouse over them.
- **Animated Graph Navigation**: Graph navigation is supported by the ability to choose new root nodes. When a new root node is chosen, the visualization displays the animation between the old and new layouts. Node positions are animated by linearly interpolating the

polar coordinates of the nodes. Since nodes can change levels between layouts, their size change is also animated.

- **Node Tooltips**: Used to display extra metadata about each node.
- **Display Properties**: Edges that don't belong to the spanning tree may be displayed or not. If they are displayed, they are made thinner.

3.2.4 Summary

The interaction techniques are quite varied across the reviewed literature. A key takeaway about these techniques is that they sit on top of layouts where information is not displayed equally. In some cases, information is hidden either to keep clutter to a minimum or because only a sub-graph is actually being drawn, while in others, it is given less importance to bring more attention to other areas (e.g. node size distortion in MoireGraphs). Some of these interaction techniques offer ways to work around this.

In the context of hidden information, tooltips are seen being used to display extra information about nodes that would otherwise clutter the graph, like is seen on the Radial+Force-directed approach and in MoireGraphs. The problem of only a sub-graph being displayed is addressed by providing ways of causing the layout to dynamically change, usually through sub-tree collapse or expansion like the widget in the Dual Trees prototype or the root node switching on the Radial+Force-directed approach. These layout transitions are portrayed through animations to help the user keep context. Dynamic layout changes also occur in MoireGraphs when switching the focus (root) node. Here there is no collapse or expansion of sub-trees but an animated rearrangement of node positions and sizes. These features are very useful to help keep the visualizations clean and help the user not lose track of things. However, some caution must be taken. Tooltips are useful to display extra information but not all of it. And animations run the risk of being too complex to follow once they start involving too many nodes.

Other interaction techniques that are worth mentioning here relate to ones that allow the user to manipulate layout algorithm parameters, like the cone steepness slider in the Radial+Force-directed approach and the focus strength and radial rotation options in MoireGraphs. Although these serve functional purposes, users might end up using them to tune the visualization to their liking. The drawback behind this is that these options should come with good defaults to save users the effort of having to tune the visualization every time.

Chapter 4

Problem and Proposed Solution

This chapter presents some insight into the challenges of drawing graphs, including Family Trees and UML class diagrams, as well as some discussion on how CleanGraph's graph interaction can be improved. It also contains some decisions made on how these improvements are going to be implemented.

4.1 Graph Layout

This section covers the static layout portion of the work. Some challenges are presented, along with existing solutions in Cytoscape.js. Finally, some decisions are made for the algorithms to be implemented.

4.1.1 Challenges of Drawing Graphs

The main aim of automatic graph layout algorithms is to create a mapping of a graph to coordinates while following a set of constraints or objectives which may or may not be imposed by the data in the graph. Some of these objectives usually involve the optimization of certain aesthetic criteria, which measure mathematically defined properties of the graph layouts. Examples of aesthetic criteria are, quoting from Eiglsperger et al. [Eiglsperger et al., 2003] [Battista et al., 1998]:

- minimize number of edge crossings `CROSSING`
- minimize number of bends `BEND`
- minimize number of node and edge overlap `OVERLAP`
- maximize number of orthogonal edges `ORTHOGONAL`
- maximize angular resolution `RESOLUTION`
- minimize edge length `EDGE LENGTH`
- minimize area `AREA`
- maximize rectangular aspect-ratio `ASPECT RATIO`
- maximize number of edges respecting flow `FLOW`

- maximize symmetry `SYMMETRY`

Studies were done on the influence these criteria have on the readability of diagrams [Purchase, 1997] [Purchase et al., 1997]. The experiments were based on students answering questions about graph drawing. Between `BEND`, `CROSSING`, `RESOLUTION`, `ORTHOGONAL` and `SYMMETRY`, `CROSSING`, `BEND` and `SYMMETRY` proved to be the most important.

It is also worth pointing out that certain criteria contradict, e.g. `CROSSING` and `AREA` contradict, since edges crossing might be solvable by routing edges through alternative paths, which leads to the graph requiring more space. As such, drawing graphs can be seen as solving a multi-objective optimization problem [Eiglsperger et al., 2003] [Battista et al., 1998].

4.1.2 Challenges of Drawing Family Trees

In essence, family trees are directed graphs that describe the relationships between people in a family. Such graphs can also be called genealogical graphs. Drawing such graphs has many challenges, some of which are mentioned in [Martins, 2021]. In an ideal situation, each individual in a genealogical graph would only marry at most once, with a non-blood-related person, and have at most two parents. However, in reality, many unusual cases can occur: people can marry multiple times, marriages can happen between blood-related people, and inter-generational couples can form, to name a few. These situations can cause constraints in automatic layouts, but it is still information that needs to be displayed.

[McGuffin and Balakrishnan, 2005] presents some analysis of the theoretic properties of genealogical graphs. The challenge of drawing these is reliant on the scheme used to represent them as well as certain properties that can be observed inside the family. If to the standard familiar relationships of *parent*, *child*, *ancestor* and *descendant*, we add the relationships:

- **Consanguine relatives**: individuals with a common ancestor.
- **Conjugal relatives**: individuals connected by an undirected path through one or more marriages, e.g. brothers-in-law.

We can define different types of intermarriage in a family:

- **Type 1 Intermarriage**: marriage between people who are consanguine relatives.
- **Type 2 Intermarriage**: marriage between people who are conjugal relatives via a path that doesn't include their marriage.

Depending on the non-existence of these kinds of intermarriages and the scheme of the graph, a genealogical graph can be considered a tree. Trees are planar and can be easily drawn without edge crossings. However, it isn't uncommon for users to require nodes to be ordered by time in a way that makes generations apparent. Such a constraint leads to edge crossings becoming unavoidable. Relaxing the constraints so that the ordering only applies between parent and child lets the edges

crossing disappear, with the drawback of very long edges [McGuffin and Balakrishnan, 2005]. If these properties can't be observed, if the scheme allows the graph to be a DAG, it can be drawn with generic drawing algorithms like Sugiyama et al.'s [Sugiyama et al., 1981], or the one in [Gansner et al., 1993].

Most commercial solutions for genealogical graph visualization address these challenges by displaying portions of the graph and hiding information to achieve a cleaner view. However, it also creates the possibility for misinterpretation of data [Martins, 2021]. Also, one of *CleanGraph*'s standout points is the representation of graphs in their entirety, so these approaches are not an option, at least for the static representation.

In conclusion, to develop an algorithm to layout genealogical graphs, several aspects can be considered, mainly the existence of intermarriage and the degree to which a user wants to retain generational ordering. The developed algorithm might allow for automatic detection of these situations and pick the best way to draw the graph. However, caution might be necessary. Users might be thrown off by this behavior, and deem the program inconsistent.

4.1.3 Challenges of Drawing UML Class Diagrams

UML class diagrams represent the architecture of a system, which means that the complexity of these diagrams is directly related to the complexity of the system itself. This complexity can be identified by an increase in the size of the diagram, be it the number of classes or the number of relations between classes. With the increase in the size of the diagram, the challenges related to drawing it become apparent.

UML class diagrams have also been studied, with some aesthetic criteria being defined for this specific type of diagram. Aesthetic criteria more specific to class diagrams are [Eichelberger, 2002] [Eiglsperger et al., 2003]:

- use hyperedge notation for generalization relation `HYPEREDGE`,
- center n-ary association `CENTER`,
- place comment nodes and association classes near to the related model elements `PROXIMITY`.

Purchase et. al also conducted research into these aesthetic criteria [Purchase et al., 2001], it concluded that `BEND` and `FLOW` worsen the readability of class diagrams, while `ORTHOGONAL` had no influence. User preference is also indirectly linked to the readability of a class diagram as different people can interpret things differently [Eiglsperger et al., 2003]. In a study on user preference for criteria in class diagrams by Purchase et al. [Purchase et al., 2000], `CROSSING`, followed by `BEND` and `HYPEREDGE` is the order of importance attributed to the respective criteria.

Eiglsperger et al. conclude from these studies that `CROSSING` and `BEND` play an important role in class diagrams. `FLOW` and `HYPEREDGE` might play an important role as well, but there isn't sufficient investigation. Nonetheless, they should be considered [Eiglsperger et al., 2003].

An algorithm that aims to draw UML class diagrams has to take into account these various aesthetics criteria. Which ones to pick and optimize is a choice of the creators. Some degree of user preference should be considered and influence the way the layout is calculated. An example of such is the work from Eiglsperger et al. [Eiglsperger et al., 2003], their layout algorithm permits the user to enforce certain edges to follow the `FLOW` criterion, e.g. drawing sub-classes below their super-classes, and the `HYPEREDGE` criterion.

4.1.4 Cytoscape.js Layout Algorithm Implementations

CleanGraph is built on top of Cytoscape.js, a graph theory library than enables graph visualization and analysis. Some of its features include: support for many graph types, graph manipulation, many graph theory algorithms, and, most notably, automatic layout algorithms [Franz et al., 2016]. Since most of this work involves improving CleanGraph's current static layouts, it is worth looking into the available options.

Cytoscape.js has many layout offerings, whether it be through native implemented ones or through extensions. A few of them are as follows [Cytoscape.js, 2021]:

- Geometric Layouts: Organize the graph into geometric shapes
  - `grid`: organizes the nodes in a well-spaced grid.
  - `circle`: organizes the nodes into a circle. Nodes are sorted manually.
  - `concentric`: places node in concentric circles. Which circle a node is placed into is based on a metric. The higher the metric, the innermost the node will be placed.
  - `avsdf`: same as `circle`. However, the nodes are sorted automatically to avoid edge overlap.
- Hierarchical Layouts: Good for trees and DAGs
  - `dagre`, `elk-layered`, `elk-mrtree`, `klay`: all very similar, `dagre` and `elk-layered` offer similar results for smaller data-sets, `klay` is a predecessor to `elk-layered`. The elk algorithms are all bundled into a single file, so the size can be a limiting factor.
  - `breadthfirst`: organizes nodes in levels according to the levels generated by a breadth-first search applied on the graph.
  - `concentric`: can be hierarchical if the metric is the level obtained from a breadth-first search.
- Forced-directed Layouts: in general, all work in a similar way. Differences might be performance or the ability to set constraints.

In conclusion, there is a wide selection of algorithms one can choose from. Hierarchical layouts are promising for genealogical graphs. However, they don't offer much customizability in terms of defining custom ways to rank nodes or ordering them within ranks. This is important

because genealogical graphs can have some semantic constraints when creating drawings of them, e.g. keeping siblings close together. They might give satisfactory results for smaller graphs, but on larger data-sets, these rules won't be as respected. So specializing these algorithms so they scale better for larger graphs is necessary. Another point is that none of the offered solutions seem to fit the layout of UML class diagrams well enough. Finally, these algorithms treat every node in the graph as equal, which is something that isn't necessarily true for Family Trees and UML class diagrams. Some nodes within these types of graphs need to be treated differently and may even be just auxiliary nodes that shouldn't be displayed. With this in mind, the algorithms to be used will need to be developed from scratch.

4.1.5 Algorithm Choices

Having discussed some of the challenges of drawing both kinds of graphs, some decisions about the algorithms to be implemented are made.

4.1.5.1 Genealogical Graphs

For genealogical graphs, as stated in Section 4.1.2, it would be ideal for the application to detect certain structural properties of the graph and use the most appropriate algorithm to efficiently draw the diagram, all while listening to the user's preference in generational ordering. However, this would require the design of multiple different algorithms, which, in the time frame and context of this work, is unfeasible. So, the algorithm to implement will be a more general solution that is still optimized for genealogical graphs. The approach in [Mařík, 2016], with its modifications to the algorithm in [Gansner et al., 1993], is a very compelling one. There is also an opportunity for improvement of the approach since the author only used their methods to define a constrained input for the algorithm and didn't implement the algorithm from scratch. Because of this, the algorithm can ignore the proposed ordering if it finds a better one, leading to wasted computing power. Directly implementing the changes into the algorithm allows for the mitigation of these. Also, since the approach aims to keep families clustered, the algorithm can be further optimized to work by reordering the clusters instead of reordering individual nodes. Another opportunity for improvement is the combination of different approaches, like in [Keller et al., 2010], the application of a force-directed approach to order the nodes within levels might be interesting, especially if it is specialized for genealogical graphs.

4.1.5.2 UML class diagrams

For UML class diagrams, there are many solutions using hierarchical methods to draw these, relying on the initial layout of inheritance relationships as a base from which to draw the rest of the diagram around. However, there are some sound arguments against the usefulness of these methods. These approaches assume that there is a large number of inheritance relations in the diagram and that these relations should be strictly drawn in an upward direction. However, in practice, this isn't necessarily the case. Class diagrams don't have to contain inheritance relations. Also,

extensive use of inheritance is actually a discouraged practice in software engineering for a long time; composition being often encouraged instead [Gamma et al., 1995]. So, there it is expected that many class diagrams have more associations than inheritance relations. Even diagrams with inheritance may suffer, as they end up not being as deep as desired for good layouts, resulting in very wide layouts. These arguments are presented in [Eiglsperger et al., 2003] along with the topology-shape-metrics approach to drawing these diagrams, which takes into consideration of the overall topology of the diagram. It also attempts to draw inheritance relation in an upward direction but doesn't force it if it sacrifices readability, taking it more as a relaxed constraint. This approach seems like a better method for drawing class diagrams.

Another thing that should be considered is the semantics behind the diagrams. The systems they represent can usually be divided into modules of closely related classes, and making sure these classes are drawn closer helps people understand the overall components of a system. The work in [Hu et al., 2012] attempts something like this by clustering the graph around a set of most important classes, with trade-offs in readability. So, it is evident that ensuring the closeness of groups of classes might conflict with readability criteria. A solution for this might be having a second layout that focuses more on grouping closely related classes.

4.2 Graph Interaction and Navigation

In this work, graph interaction and navigation pertain to how a user can interact with graph visualizations to better extract information from them and build a better understanding of the data. Over the analyzed state of the art, many interactive features and methods were observed. One thing which can be taken away from it is that these interaction methods are centered around the visualization of subsets of the graph, focusing on a certain part of the graph or a mixture of both.

Interaction based on partial graph visualization can be tied to how these works choose to deal with scale. As graphs get bigger, these interaction methods manage scale by reducing the amount of data displayed at once and provide features that allow users to quickly and effectively navigate and extract information. These become less needed when full graph visualizations are used since all the information is displayed at once and not hidden. The challenge with this approach centers more around keeping the graph layouts clean and semantically readable, which is explained in the section above. This does not mean that full representation of graphs lack of interaction features, just that they are less interesting in a sense.

Interaction with full graph visualizations has the aim of allowing a user to extract information more easily through, for example, allowing for changes in the level of detail of the graph's elements or helping the users spot elements more easily. One very basic feature is zooming and panning [Herman et al., 2000]. Using these, the user can view certain parts of the layout with more detail and focus on those. Another is node highlighting, which helps the user in spotting nodes that match certain criteria. Pairing this with a search feature can be very useful. Lastly, enabling the user to manually rearrange the layout by moving nodes to different positions allows for better productivity as the initial drawing may have some shortcomings and not fit the user's preferences

[Keller et al., 2010]. These are all things that CleanGraph already implements in its interaction model, some of which are enabled by the use of Cytoscape.js [Martins, 2021].

That being the case, the improvement of CleanGraph's graph interaction might actually rely on the use of secondary layouts and interaction techniques like the ones found in the state of the art. This means that these layouts won't be focused on the full graph visualization. They will instead show parts of graphs or emphasize certain nodes. Traversal of these layouts will then have to rely on well-crafted animated transitions to help the user maintain context. An example would be the Dual Trees visualization [McGuffin and Balakrishnan, 2005], or the MoireGraphs [Jankun-Kelly and Ma, 2003] which could be applied to UML class diagrams to help a user better understand the relations between classes.

4.3 Completed Work

This section now covers the work that was completed in the scope of this dissertation. This work is divided into three main tasks: implementation of a new Family Trees layout algorithm, implementation of a new UML class diagrams layout algorithm, and the update of the existing navigation feature for UML class diagrams.

The Family Trees layout algorithm is based on the hierarchical style of layout algorithms and follows the specification of existing work. For this, some alterations in the graph representation of Family Trees were necessary for the graph to be compatible with the algorithm.

The UML class diagrams layout algorithm is based on the Topology-Shape-Metrics approach, and its implementation also follows the specification of existing work. Like in Family Trees, the graph representation needed to be altered for the graph to be compatible with the algorithm.

The navigation feature update was based on the implementation of a new radial layout of UML class diagrams, which allows the user to jump from class to class and allows them to analyze the relationships between classes from a different perspective.

The next chapter will contain implementation details for each of the tasks performed.

Chapter 5

Implementation

This chapter is going to cover the implemented layouts for each graph type, i.e. Family Trees and UML Class Diagrams, explaining how they work with implementation details, along with documenting changes that were done to the existing work to accommodate the new layouts.

5.1 Family Trees Layout

In family trees, the relationships between parents and children are hierarchic in nature. Hierarchical graph drawings are very good at representing flow in DAGs. If in Family Trees, we consider the flowing "thing" to be time or the passing of lineages onto new generations, these kinds of drawings can be very useful for representing Family Trees, as already seen in the work analyzed in previous sections.

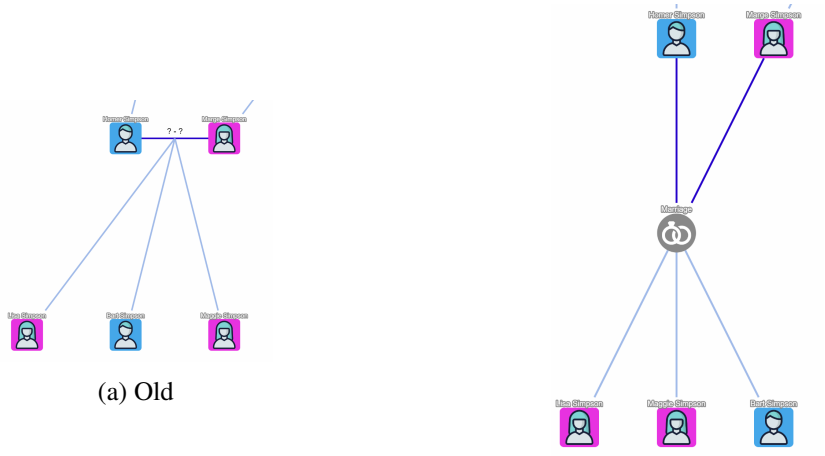
As mentioned in previous chapters, hierarchical graph drawings try to assign nodes into hierarchies so as to ensure that every edge points in an upward direction. They generally have three steps:

1. **Layering –** Assign nodes to layers.
2. **Node Ordering –** A linear order in each layer is computed in order to minimize the number of edge crossings.
3. **Coordinate Assignment –** Assigning coordinates to nodes in order to optimize certain aesthetic criteria.

The implemented algorithm for Family Tree layout is mostly based on Gansner et al's work [Gansner et al., 1993]. The following sections explain implementation details of the algorithm.

5.1.1 New Representation

Before implementing the layout algorithm, some changes to the existing representation of Family Trees need to be made. The current version of CleanGraph represents marriages as edges, as



(a) Old

(b) New

Figure 5.1: Same marriage represented with the old and new graph representations.

they are essentially a relationship between two people (nodes). However, this results in children resulting from marriages being represented as edges connecting the children to the marriage edge. Traditional graphs don't support the idea of edges connecting to other edges, meaning that layout algorithms also don't work over this kind of representation. In light of this, marriages were converted from edges to nodes. A more formal explanation follows: every edge $m = (v, w)$ representing a marriage between persons v and w, and respective edges $(c_1, m), \ldots, (c_i, m)$ representing the children $c_1, \ldots, c_i$ born from said marriage, are replaced with a node M and edges (v, M), (w, M) and $(M, c_1), \ldots, (M, c_i)$.

This can be seen as marriages being less treated as a relationship and more as an event. And the direction of the edges connected to marriage nodes represents the role each person has in that marriage. Figure 5.1 shows a comparison between both representations.

5.1.2 Layering

As mentioned above, this step involves the assignment of nodes to layers. The center objective is to ensure that every edge in the graph points in the same direction, i.e. pointing from a lower layer to a higher one, or vice versa. This is only possible if the graph is acyclic, so an important part of this step in general layout algorithms is ensuring that the graph is a DAG, and if it isn't, transform it into one by reversing some edges, this step is trivial if no constraints are applied on it, e.g. reverse the minimum amount of edges.

In the case of Family Trees in CleanGraph, the graph is guaranteed to be a DAG. The transformation in the section above ensures the edges flow from parent to child in marriages, and the GEDCOM[1] to graph conversion already ensures that direct relationships between parent and child point from the parent to the child. This eliminates the possibly costly first part of this step.

[1] File format for exchanging genealogical data: https://www.gedcom.org/.

The actual assignment of layers to nodes can be done through a simple random walk algorithm, starting at a node at layer 0 and traversing through the graph along the edges, assigning layers to nodes depending on the direction the edge was traversed. If the traversal flowed from a child node v to a parent of marriage node w, w's layer $l(w)$ will be $l(v)-1$. If the traversal flowed from a parent node v to a marriage or child node w, then $l(w) = l(v)+1$.

However, some constraints were applied to the basic algorithm. Since marriages are now represented as nodes, the graph now has two types of nodes, representing two distinct things. It might be undesirable to have person nodes and marriage nodes next to each other as it can cause visual clutter. To ensure this, the algorithm assigns person nodes to even layers and marriage nodes to odd layers. This way, person nodes, and marriage nodes are assigned to distinct layers and can be more easily differentiated.

5.1.3 Node Ordering

The node ordering step of the algorithm is focused on creating a linear order of the nodes in each rank so that the overall number of edge crossings in the whole graph is minimized. The implementation of the algorithm follows the method in Gansner et al's work [Gansner et al., 1993]. Algorithm 1 shows the general structure. A set number of iterations are run over the entire graph. Each iteration runs the *wmedian*() heuristic (Algorithm 2), which sorts the nodes in each layer according to the median positions of their neighbors in adjacent layers. Depending on whether the iteration is odd or even, the heuristic puts a bias on the positions of neighbors in higher or lower layers, respectively.

Algorithm 1: Node Ordering [Gansner et al., 1993]

```
1 order ← initial_order();
2 best ← order;
3 for i in 0..max_iter do
4 |   wmedian(order, i);
5 |   if crossing(order) < crossing(best) then
6 |   |   best ← order;
7 |   end
8 end
```

Algorithm 3 shows how the median position of a node's neighbours is calculated. This median is calculated as a weighted median value, the objective is to influence nodes to be placed closer to areas where it's neighbours are more closely packed. *adj_positions*() in line 1 returns a sorted list of the positions of node v's neighbours in the layer *adj_rank*. If a node does not have neighbours in *adj_rank*, the algorithm will make sure its position doesn't change.

The sorting algorithm, specified in Algorithm 4, sorts nodes based on the calculated median position of their neighbours, nodes marked with a median of -1, i.e. nodes without neighbours, are kept in their original positions.

Algorithm 2: *wmedian* Procedure [Gansner et al., 1993]

```
   Data: order, iter
 1 if iter mod 2 = 0 then
 2 |   for r in 1..Max_rank do
   |   |   /* Sort nodes in ranks based on positions of neighbours
   |   |      in the previous layer                                  */
 3 |   |   for v in order[r] do
 4 |   |   |   median[v] ← median_value(v, r − 1);
 5 |   |   end
 6 |   |   sort(order[r], median);
 7 |   end
 8 else
 9 |   for r in (Max_rank − 1)..1 do
   |   |   /* Sort nodes in ranks based on positions of neighbours
   |   |      in the next layer                                      */
10 |   |   for v in order[r] do
11 |   |   |   median[v] ← median_value(v, r + 1);
12 |   |   end
13 |   |   sort(order[r], median);
14 |   end
15 end
```

Algorithm 3: *median_value* Procedure [Gansner et al., 1993]

```
   Data: v, adj_rank
 1 P = adj_positions(v, adj_rank);
 2 m = |P|/2;
 3 if |P| = 0 then
 4 |   return −1.0
 5 else if |P| mod 2 = 1 then
 6 |   return P[m]
 7 else
 8 |   if |P| = 2 then
 9 |   |   return (P[0] + P[1])/2
10 |   else
11 |   |   left ← P[m − 1] − P[0];
12 |   |   right ← P[|P| − 1] − P[m];
13 |   |   return (P[m − 1] ∗ right + P[m] ∗ left)/(left + right)
14 |   end
15 end
```

Algorithm 4: *sort*() Procedure

```
   Data: rank, median
 1 F ← [];
 2 for i in 0..|rank| do
 3 |   if median[rank[i]] = −1 then
 4 |   |   F+ = (rank[i], i);
 5 |   |   Remove rank[i] from rank;
 6 |   end
 7 end
 8 Sort rank based on median values;
 9 for (v, i)inF do
10 |   Insert v into rank at index i;
11 end
```

5.1.4 Coordinate Assignment

The coordinate assignment step of the algorithm calculates the coordinates of the nodes while trying to optimize certain heuristics like edge length and edge slope (i.e. how "vertical" they are).

Y-coordinates are calculated based on the layer the nodes were assigned.

X-coordinate calculation (Algorithm 5) follows a similar approach to the node ordering step. It also runs a set of iterations over the graph. At each iteration, it runs a heuristic that places each node at the median position of its neighbors in adjacent layers. Depending on the iteration, the neighbors in the upper or lower layer are considered, just like the node ordering algorithm. The main difference is that the median is no longer weighted, as the node position that minimizes the length of a node's edges is the median position of its neighbors [Gansner et al., 1993]. Another aspect to keep in mind when placing nodes is that the node order has to be respected. So, the coordinate in which each node is placed is clamped, so it always stays to the right of nodes that come before it in the layer's linear ordering.

Algorithm 5: X-coordinate Assignment

```
1 xcoords ← initial_xcoords();
2 best ← xcoords;
3 for i in 0..max_iter do
4 |   median_pos(xcoords, i);
5 |   if xlength(xcoords) < xlength(best) then
6 |   |   best ← xcoords;
7 |   end
8 end
```

Algorithm 6: *median_pos* Procedure

Data: *coords*, iteration $iter$, node separation amount S

```
1  if iter mod 2 = 0 then
2      for r in 1..Max_rank do
3          lowerBound ← −∞;
4          for v in coords[r] do
5              m ← median_x(v, r − 1);
6              lowerBound ← max(x, lowerBound + S);
7              median[v] ← lowerBound;
8          end
9      end
10 else
11     for r in (Max_rank − 1)..1 do
12         for v in coords[r] do
13             m ← median_x(v, r + 1);
14             lowerBound ← max(x, lowerBound + S);
15             median[v] ← lowerBound;
16         end
17     end
18 end
```

5.2 UML Class Diagrams

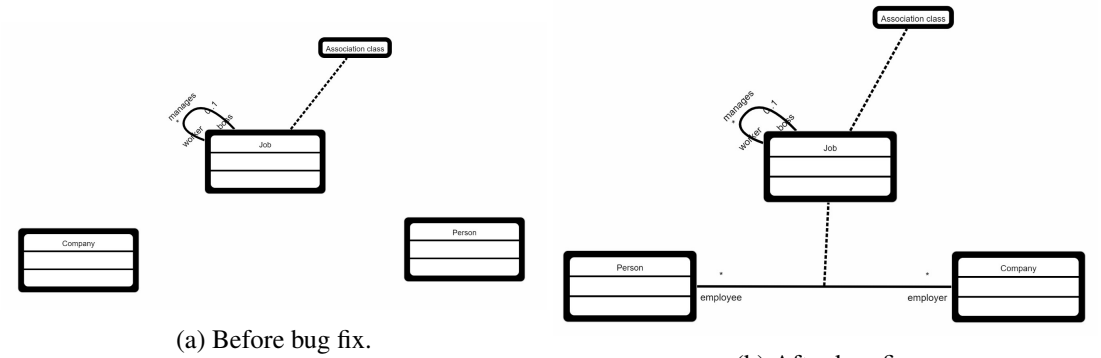
This section covers the implementations of features for the visualization of UML class diagrams. A new full graph layout algorithm was implemented, along with an update to CleanGraph's radial layout feature.

5.2.1 Global Layout

The global layout algorithm for UML Class diagrams draws the graph that represents the diagram in its entirety. The kind of drawing it tries to achieve is an orthogonal box drawing, i.e. a graph drawing where nodes are represented as boxes and edges are drawn as sequences of horizontal and vertical lines. The algorithm implementation follows the work by Eiglsperger et. al. [Eiglsperger et al., 2003]. Their algorithm intakes a UML class diagram as a mixed graph and creates an orthogonal style mixed upward planar drawing of it. The main reason the class diagram is taken as a mixed graph is so that the mixed upward planar drawing ensures certain relations between classes are drawn in the upward direction, i.e. inheritance/generalization relations, which are commonly drawn with the super-class above the sub-classes.

The algorithm models a UML class diagram as a mixed graph $G = (V, E)$, where classes are nodes V and relations between classes are edges E. There is also a subset $D \in V$ of directed edges. Recapping the explanation from Section 3.1.3.3, the algorithm works in three main phases:

- Planarization — The graph is planarized and a planar embedding of it is constructed. This is an important step because if a graph in not planar, it can't be drawn in the plane without



(a) Before bug fix.

(b) After bug fix.

Figure 5.2: Before and after the association bug fix.

edge crossings. If the graph's embedding is not planar the rest of the algorithm also won't function properly.

- Orthogonalization — An orthogonal representation of the graph is created, in which the angles and bends in the drawing are specified. This is necessary input for the next step of the algorithm. The bends and angles that are calculated during this phase are increments of 90°. This will give the drawing a orthogonal look, which usually a style UML class diagrams are drawn in.
- Compaction — In this step, node coordinates are finally calculated using the orthogongal representation as input.

Since much of the algorithm was already explained from Section 3.1.3.3 downwards. The following sections will provide some more insight into the executed steps, as well as provide some implementation details that were relevant throughout the implementation of the layout algorithm.

5.2.1.1 Association Bug Fix

Before the implementation of the algorithm, a bug in the converter responsible for turning the XMI[2] files into graph data for Cytoscape.js needed to be fixed. The issues lay in the generation of association edges. When an association had an association class, the association class would be created along with the edge to connect that class. However, the creation of the association edge itself was being skipped when this was the case (e.g. Fig. 5.2).

5.2.1.2 Pre-Processing Step

Before laying out the UML Class Diagram, some changes need to be made to the graph representation of the diagram. Like in Family Trees, there are some instances of edges connecting other edges. That is the case for associations with association classes and user-defined constraints.

[2]A standard for exchanging data, commonly as an interchange format for UML models, https://www.omg.org/spec/XMI/2.5.1/About-XMI/

These cases are converted to valid graph representations by splitting the connected edges into two segments with an intermediate node:

- For every edge $e = (v,w)$ representing an association with an association class C to which it is connecting with edge $j = (e,C)$, edges e and j are removed and replaced with an intermediate node e_I, and edges (v,e_I), (e_I,w) and (e_I,C).
- For every edge $c = (e,j)$ representing a user defined constraint between association edges $e = (v,w)$ and $j = (s,t)$, edges c, e and j are removed and replaced with intermediate nodes e_I and j_I, and edges (v,e_I), (e_I,w), (s,j_I), (j_I,t), (e_I,j_I).

About the definition of which edges are directed. All edges representing inheritance relations are marked as directed. This way, there is no need for the step of removing directed edges since inheritance relations don't cause cycles. The step that marks extra edges as direct to induce a connected graph proceeds in the same way as the specification.

Another pre-processing step deals with self-associations. Self associations create loop edges, i.e. edges connecting a node to itself, which aren't well handled. Since these edges connect twice to a node, they need to appear twice in graph embeddings. This can lead to many special edge cases along the algorithm. This also seems to be a case not handled often in literature as well as the specification, specifically for the algorithms in the planarization and orthogonalization phases. For the correct handling of these cases, these edges are temporarily removed from the graph and re-added at a later, intermediate point.

5.2.1.3 Planarization

The planarization step was implemented as specified in Eiglsperger and Kaufmann's work. As already mentioned in Section 3.1.3.4, the algorithm in this step creates a mixed upward planar embedding of the graph. This means that the output of this step is a planar embedding of the graph.

Usually, it is easy to calculate a planar embedding if a drawing of the graph is already present. However, the drawing of the graph is only obtained after the entire layout algorithm. Creating an intermediate layout just for this step would also be inefficient. Luckily, after the calculation of the maximum planar sub-graph, it is possible to use the obtained information to calculate a planar embedding for the sub-graph. During this calculation, the nodes are arranged in a line, and the sub-graph's edges are marked as being routed through the $RIGHT$ or $LEFT$ of this line. This information specifies a "virtual drawing" of the sub-graph, through which the planar embedding can be calculated as so:

1. For each node v:
   (a) Let $LEFT(v)$ be the list of nodes to which v is directly connected through $LEFT$ edges. And $RIGHT(v)$ be the list of nodes to which v is directly connected through $RIGHT$ edges.

(b) Sort $LEFT(v)$ by descending order, and $RIGHT(V)$ by ascending order.

(c) Create $l(v)$ by appending both lists together.

After this, as edges are re-added into the sub-graph, the planar embedding can be further edited to stay up to date with the graph.

5.2.1.4 Augmenting Graph into S-T Graph

The planarization algorithm requires the calculated maximum planar sub-graph to be an s-t graph. And, if it isn't, it needs to be augmented into one.

As already explained in Section 3.1.3.5, the augmentation requires the graph to be upwards planar, meaning it needs to have an upwards planar embedding. This is easy to guarantee since, if the undirected edges in the sub-graph are directed according to the calculated ordering in the planarization step, the derived planar embedding from the previous section is guaranteed to be upwards planar.

The main concern of the algorithm is the definition of the type of angles made by face switches. If a planar upward drawing of the graph was available, the angles could be determined trivially through the slope of the edges. However, as discussed above, an intermediate layout of the graph isn't efficient. To be able to deduce the angle, the information from the upwards planar embedding of the graph can be used to deduce the angles.

Considering a node v and the following sets of edges:

- $TR(v)$: the list of v's *out* RIGHT edges sorted by the descending order of the neighbour they connect to.
- $TL(v)$: the list of v's *out* LEFT edges sorted by the ascending order of the neighbour they connect to.
- $BL(v)$: the list of v's *in* LEFT edges sorted by the ascending order of the neighbour they connect to.
- $BR(v)$: the list of v's *in* RIGHT edges sorted by the descending order of the neighbour they connect to.
- $OUT(v)$: the list resultant from appending $TR(v)$ and $TL(v)$.
- $IN(v)$: the list resultant from appending $BL(v)$ and $BR(v)$.

$OUT(v)$ is a list of v's *out* edges sorted in the clockwise order, starting at the left most one. And, $IN(v)$ is a list of v's *in* edges sorted in the clockwise order, starting at the right most one. Figure 5.3 exemplifies.

We can check if a face switch forms a big or small angle by relying on the IN and OUT lists. Considering node v is a face source, and e_1 and e_2 the edges surrounding it in a certain face such that e_1 comes before e_2 in the face's clockwise order. Considering as well, that $I(e,v)$ is the index

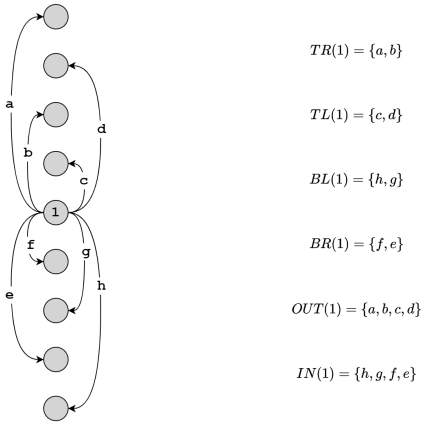


Figure 5.3: Creation of node's lists.

of edge e in $OUT(v)$. If $I(e_1) < I(e_2)$, then v is creating a big (B) angle in the face. If not, it is creating a small (S) angle. The same logic can be applied for face sinks, but in that case the $IN(v)$ list is considered. Figure 5.4 visualizes the logic.

An implementation detail about this algorithm is that the original specification only works with bi-connected graphs, meaning all faces are circuits. This means that faces where there are duplicate edge entries are not allowed. A way to adapt this algorithm to work with any kind of graph is by inducing non-circuit faces into circuit faces (Fig. 5.5). This allows for the list of symbols to be created. This is done implicitly during the algorithm by checking each pair of edges in the face traversal. If the node they share is the source or target node of both, the node is considered a face switch at that position. In Figure 5.5, node 1 is a switch of face $f1$, but only in one case.

One final point of information that can be extracted from the algorithm is the external face of the graph. Knowing which face is the external face is important for the directed edge addition algorithm since it needs to split it into two sides. This is easy to do for upwards embedded planar graphs. It is a property of these kinds of embeddings that face switches in internal faces have two more small angles than big ones, and in the external face have two more big angles than small ones [Bertolazzi et al., 1994]]. So, to find the external face of the graph, one just needs to look for the face that has two more big angles than small ones in its switches.

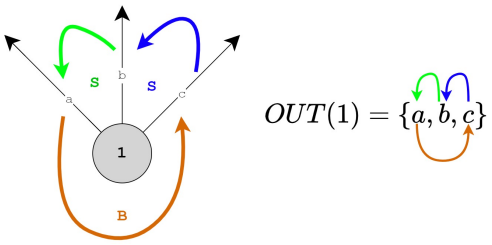


Figure 5.4: How big and small angles are determined. Taking advantage of information on upwards planar embeddings. Edges pointing in the face traversal direction.

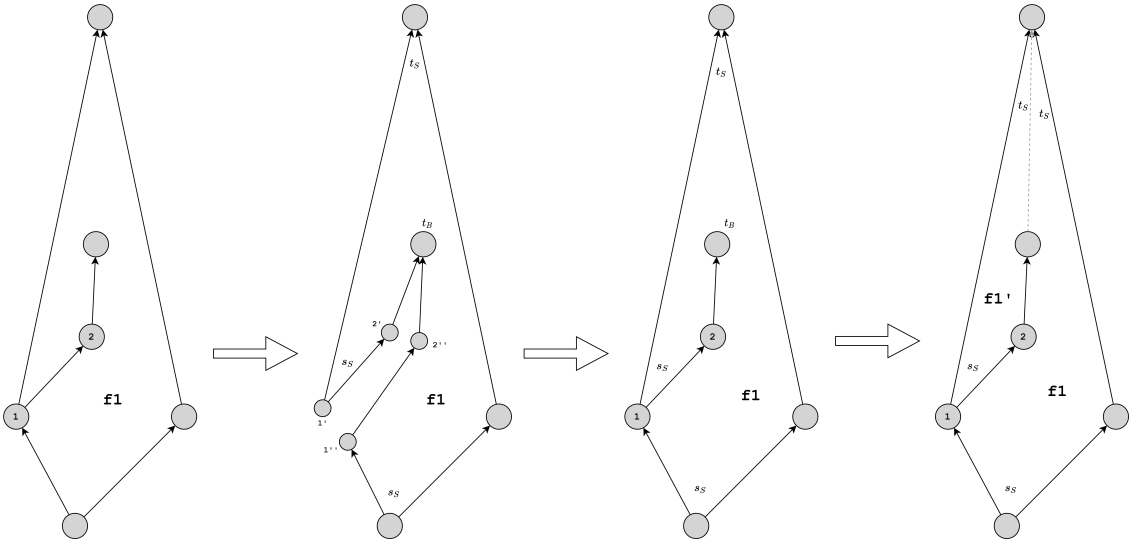


Figure 5.5: Dealing with non circuit faces in s-t graph augmentation.

5.2.1.5 Orthogonalization

The orthogonalization step in this implementation skips the edge shape pre-calculation and simply implements the algorithm by Fößmeier and Kaufmann for the computation of an orthogonal representation [Fößmeier and Kaufmann, 1996]. This means that the extension to the flow network to accept graph sketches was also not implemented. The reason for this was simply time constraints. The only effect this has is the inability to control what the graph will look like. It will still produce valid orthogonal drawings.

5.2.1.6 Self Associations

After the orthogonalization step, the self-association edges that were removed in the pre-processing step can now be re-added to the graph. Before they are re-added, the situation of the same edge appearing twice in a node's embedding list is avoided by splitting the edge into two *sub edges* connected by an intermediate node. These edges are placed such that they are rendered in the top left corner of the node with a rectangular shape (Fig. 5.6). This step is done after the orthogonalization because the current implementation of the step doesn't guarantee which shape the edge will have. So, when inserting the new edges, the orthogonal representations that were calculated will have to be altered to accommodate the new edges. Another point to note is that the addition of these two-edge cycles effectively creates a new face in the graph, for which an orthogonal representation will need to be "manually" calculated. Figure 5.7 exemplifies the process for the addition of a single self-association.

5.2.1.7 Compaction\Coordinate Assignment

The compaction phase was also implemented without deviation or extension of the work by Eiglsperger and Kaufmann [Eiglsperger and Kaufmann, 2002].

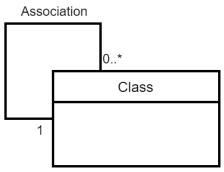


Figure 5.6: Example of how self associations are drawn.

An implementation detail worth mentioning is the definition of the A_u and A_r sets for the D_u and D_r constraint graphs. The original paper's definition assumes that all edges in the graphs are pointing either *right* or up. However, in more practical scenarios, this is not always the case, so a more general algorithm to define these sets is necessary. Thanks to the absolute directions appended to H during the preparation steps of the algorithm, it is possible to calculate which side $s = \{top, bottom, left, right\}$ of a node each edge attaches to. Algorithm 7 uses this information to calculate the sets.

Another implementation detail worth mentioning is the definition of right and left joined edges in segments, which are then used to define the sub-segments of the segment. The original paper's definition of right and left join makes the same assumptions as above. Considering a horizontal segment of the graph, a more general algorithm to identify left and right joins inside it is the following:

1. For each node in the segment, filter its clockwise list of edges to only contain edges inside the segment.
2. For every edge e in the segment:
   (a) **TO FIND RIGHT JOINS**
   (b) Find the face in which e is traversed to the *right*.
   (c) In the traversal of the face, pick the node v that follows e.
   (d) In v's filtered clockwise list of edges, pick the edge w that follows e in the anti-clockwise direction.

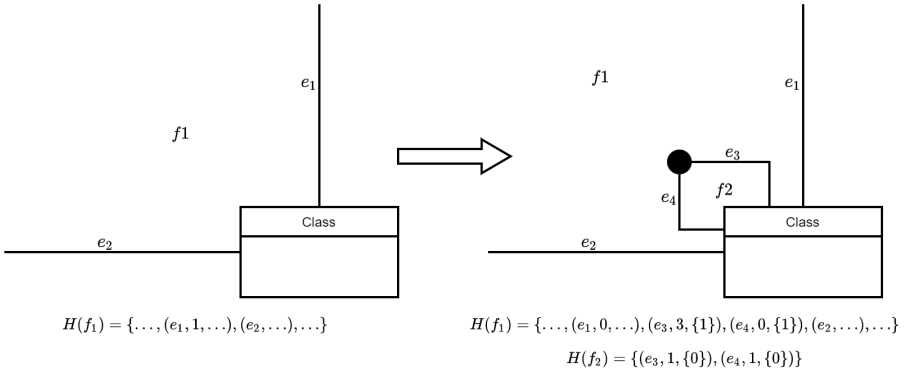
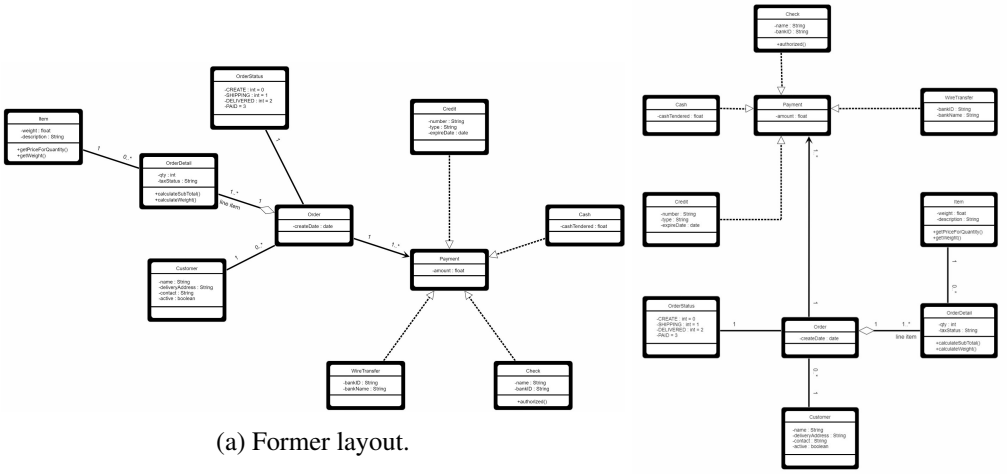


Figure 5.7: Example of insertion of split self association into orthogonal representation.

Algorithm 7: Creation of A_u and A_r sets

Data: An orthogonal representation H of the graph.

Initialize empty sets A_r and A_u;
for *Every edge* $e = (v,w)$ *in* E_u **do**
    /* Vertical edges are only attached to either the top or bottom sides. */
    $s \leftarrow$ Side of node v edge e is attached to;
    if $s = top$ **then**
        Append edge $(hor(v), hor(w))$ to A_u;
    else
        Append edge $(hor(w), hor(v))$ to A_u;
    end
end
for *Every edge* $e = (v,w)$ *in* E_r **do**
    /* Horizontal edges are only attached to either the left or right sides. */
    $s \leftarrow$ Side of node v edge e is attached to;
    if $s = right$ **then**
        Append edge $(vert(v), vert(w))$ to A_r;
    else
        Append edge $(vert(w), vert(v))$ to A_r;
    end
end
return *Sets* A_r *and* A_u



(a) Former layout.

(b) New layout.

Figure 5.8: Former and new UML class diagram layouts.

(e) If w is attached to the opposite side of v that e is attached to, v and w are right-joined.

(f) **TO FIND LEFT JOINS**

(g) Find the face in which e is traversed to the $left$.

(h) In the traversal of the face, pick the node v that follows e.

(i) In v's filtered clockwise list of edges, pick the edge w that follows e in the anti-clockwise direction.

(j) If w is attached to the opposite side of v that e is attached to, v and w are left-joined.

To find left and right joins in vertical segments, the same algorithm can be run. However, instead of looking for the faces where edges are traversed to the right or left, the searched faces are the ones where edges are traversed up or $down$.

5.2.1.8 Results

The results of the implementation seem satisfactory. The former layout solution worked by picking the class with the highest amount of relations to other classes, placing in the center. After that, the algorithm would try to divide the area around the center class for the other classes around it. The final result was a radial like layout (Fig. 5.8a) [Martins, 2021].

The new implementation follows the orthogonal aesthetic, creating drawings that resemble the more traditional UML class diagram look (Fig. 5.8b).

5.2.2 Radial Layout

This section covers further work on a feature of CleanGraph initially implemented in previous work. The radial layout [Martins, 2021]. This radial layout is part of a feature of the former global

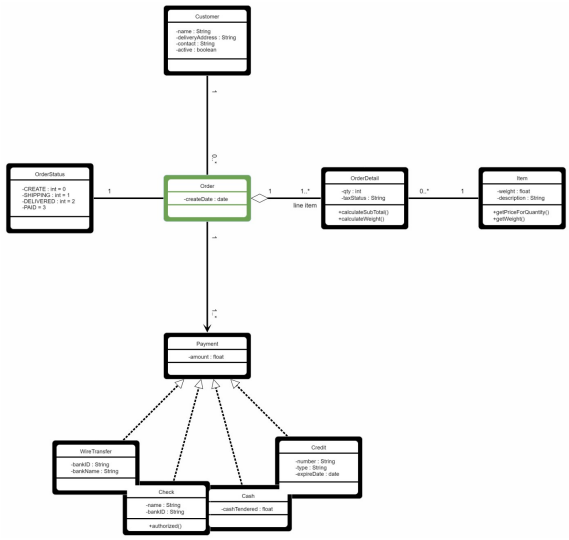


Figure 5.9: Example of former Radial UML layout.

layout that allows the user to select a different center class to create the drawing around (Fig. 5.9). Now, this feature is updated with a new implementation and some extra diagram styling features.

After being presented with the fully drawn graph, the user is then able to select a class in the diagram. Selecting a class will create a temporary radial layout that displays the selected class in the center, with the remaining classes arranged around it. The layout takes inspiration from MoireGraphs [Jankun-Kelly and Ma, 2003]. Classes are arranged in layers around the center.

The layer each class belongs to is based on a spanning tree. The spanning tree is calculated using a BFS starting from the selected class. The layers of each node are calculated based on the number of hops along the tree the node is far from the center. From this calculation a data structure can also be created: considering a node $v \in V$ of the graph G, $l(v)$ represents the layer of node v, and $C(v)$ represents the children of v, i.e. the nodes in layer $l(v)+1$ that are connected to v through an edge of the spanning tree.

The coordinates of the nodes in the radial layout are determined with polar coordinates (d, θ), which are calculated by the radial layout algorithm. To be able to determine these coordinates, three things need to be calculated:

1. The node radius of each layer.
2. The radius of the layer.
3. The angular span of each node.

The node radius of each layer represents how much space is occupied by the nodes within it. It must be big enough to fit any of its nodes without any "spill". The MoireGraphs solution sets an initial size for the first layer, making subsequent ones smaller by a certain fraction. This has in mind that all nodes have the same size, and its goal is to make nodes appear smaller as they get farther away from the center, so they are scaled down to fit the layer anyway. In the case of UML

classes, it is not desirable to have the nodes scale down, adding the fact that Cytoscape.js doesn't provide facilities to do that easily. The node radius of each layer is determined based on the size of the nodes within it. Given a layer l_i, its node radius r_{n_i} is equal to the length of the diagonal of the node with the biggest diagonal in l_i.

The radius of each layer represents how far away the layer is from the center. The radius of a layer l_i depends on the radius of the previous layer plus its node radius. Given layers $l_0, l_1, \ldots, l_k$ with node radii $r_{n_0}, r_{n_1}, \ldots, r_{n_k}$, the radii $r_{l_0}, r_{l_1}, \ldots, r_{l_k}$ of the layers are calculated as such:

$$
\begin{aligned}
r_{l_0} &= 0 \\
r_{l_1} &= r_{n_0} + r_{n_1} + P \\
r_{l_i} &= r_{l_{i-1}} + r_{n_{i-1}} + r_{n_i} + P, i \in [2,k]
\end{aligned}
$$

P is a padding constant that can be adjusted to add more spacing between layers. The radius of the layer a node is inside maps directly to the d portion of its polar coordinate.

The angular spread of a node relates to how much angular space it takes up in a radial layout. A node's angular spread depends on the angular spread of its children, the angular spread of its siblings, and the angular spread of its parent. First, a bottom-up pass is done, starting at the leaves of the tree and moving up towards the root node. A node's angular spread is only calculated when the angular spread of all its children is calculated. If a node v is a leaf of the tree, its angular spread $\alpha(v)$ is calculated with $\alpha(v) = 2 \times arctan(r_{n_i}/r_{l_i})$, with $i = l(v)$. If v is not a leaf, $\alpha(v) = \sum_{w \in C(v)} \alpha(w)$. After this pass, all nodes will have an angular spread calculated, even the root node n_r. However, it is not guaranteed that sum of the angular spreads of the children of the root node n_r totals at 2π radians. To address this, a second, top-down pass is performed to readjust the angular spreads of nodes. The angular spread of n_r is forcibly set to 2π, then its children's angular spreads are re-scaled to fit the new angular spread. This might result in them becoming smaller or larger. However, their proportions are kept. This re-scaling is then performed recursively down the list. Taking a node n with new radial spread $\alpha(n)$, the sum of its children's current angular spreads is calculated $ChildSum = \sum_{w \in C(v)} \alpha(w)$, for each children $c_i \in C(v)$ its new angular spread is equal to $(\alpha(c_i)/ChildSum) \times \alpha(v)$. The angular spread of all nodes, except for n_r, is limited to at most π radians. This avoids cases of root nodes with only one child then being surrounded by their grandchildren, etc.

The final placement of the nodes is based on polar coordinates. The angle portion is based on the angular spread. Nodes are place in the center of their assigned region. The radius portion of the coordinates is based on the radius of the layer where the node is placed. Algorithm 8 specifies the process.

Edges in the radial layout have different styles depending on whether they are part of the spanning tree or not. All edges which are not part of the spanning tree are grayed out in order to highlight the ones that are part of it. This helps with identifying the relation path between the center class and the other ones. The nodes representing classes in the diagram are also styled

Algorithm 8: placeNodes(), Radial Coordinate Assignment

Data: Node v
Data: Starting angle θ
Data: Center position (x, y)
1 $angle \leftarrow \theta + (\alpha(v)/2)$;
/* $r(l(v))$ = radius of the layer in which v is placed */
2 $x(v) \leftarrow cos(angle) \times r(l(v)) + x$;
3 $y(v) \leftarrow sin(angle) \times r(l(v)) + y$;
4 $currentAngle \leftarrow \theta$;
5 **for** $w \in C(v)$ **do**
6 $\quad placeNodes(w, currentAngle, centerPos)$;
7 $\quad currentAngle \leftarrow currentAngle + \alpha(w)$;
8 **end**

according to the layers they are on. The further away they are from the center class, the more transparent they become. This is made in the hope of helping highlight the degree of importance each class has in relation to the class in the center. Figure 5.10 shows an example of the new radial layout on the same graph as Figure 5.9 with the same class at the center.

After the radial layout is complete, if the user selects another class, the layout is recomputed with that class at the center. The nodes' positions are linearly interpolated from their old positions to the new ones over half a second. This interpolation also happens when going from the global layout to the radial one. The styles of the components in the graph, i.e. edge color and node opacity, also get recomputed and transitioned smoothly.

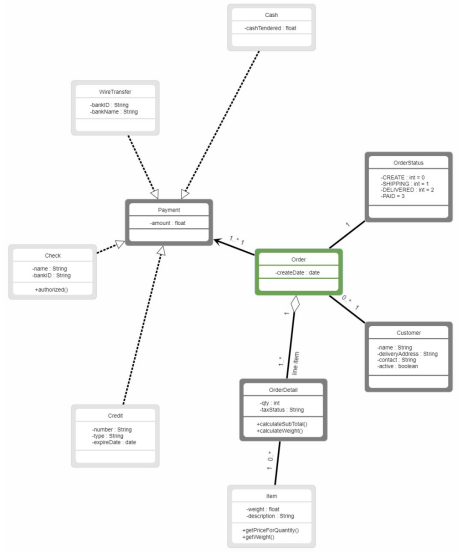


Figure 5.10: Example of new Radial UML layout.

Chapter 6

Discussion

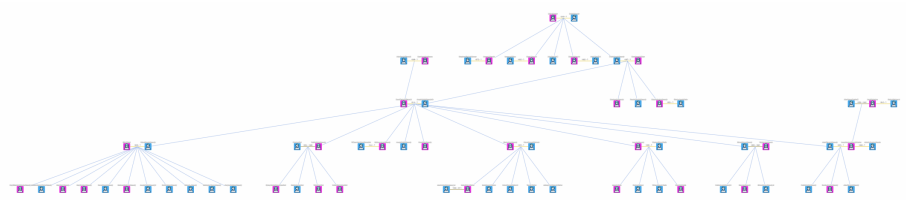
This chapter will cover the results of the implementation process and compare them to the previous solutions. Comparisons will be made between the old and new solutions. Some possible pros and cons of each solution will be presented. Finishing off with possible paths to take to improve the layouts or implement more features in them.

6.1 Family Trees

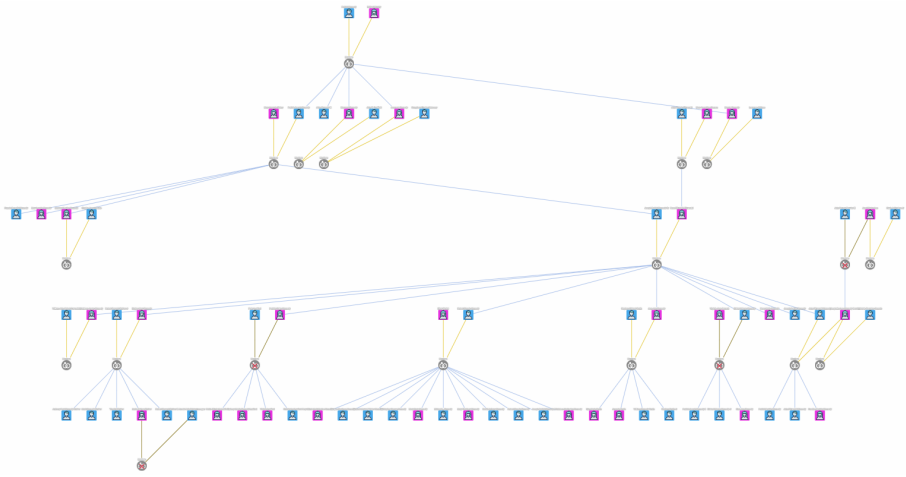
The conversion of marriages from edge relationships to nodes increases the node count of the graph considerably. This increase in nodes also increases the number of layers they are distributed into. This has the effect of increasing the height of the graph. Figure 6.1 shows a comparison of both algorithms' hierarchical style layouts of the same tree.

The new marriage nodes can help provide clarity on identifying families due to the bundles of edges forming around them. It can also be a better option than the old layout when people have more than two marriages. The switch from edge to node also helps in extracting more information from the graph. Previously, all of the marriage information was displayed in a tooltip that would appear when the marriage edge was hovered. Now, since marriages are nodes, they are easier to target with a mouse or tap on a touchscreen, and their information is now displayed in the auxiliary bar. Marriages are also easier to distinguish from divorces. When using edges, they were differentiated using the color of the edge. Marriages nodes are differentiated from divorce nodes through their shape with a different icon.

CleanGraph has a feature that modifies the layout to use the vertical axis to represent the birth year of people. In this layout, the people's ages are better represented than their generations. Since this layout is created by modifying the initial one, the new layout can also use this same secondary layout. The only necessary modification was to provide a value for the marriage nodes to be interpolated, i.e. the date of marriage. In this aspect, the old layout seems to fair better with nicer drawings. Figure 6.2 shows a comparison. A possible reason seems to be that the addition of marriage nodes makes the graph more cluttered. Since children tend to be born soon after marriages, the overlap between both nodes might happen (Fig. 6.3a). Another situation that might



(a) Old layout.



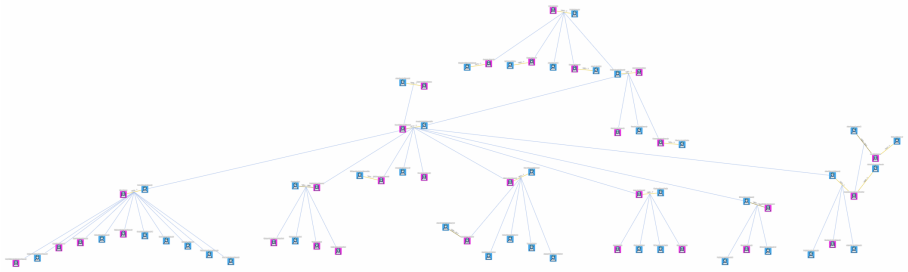
(b) New layout.

Figure 6.1: Comparison of the old and new family tree layouts.

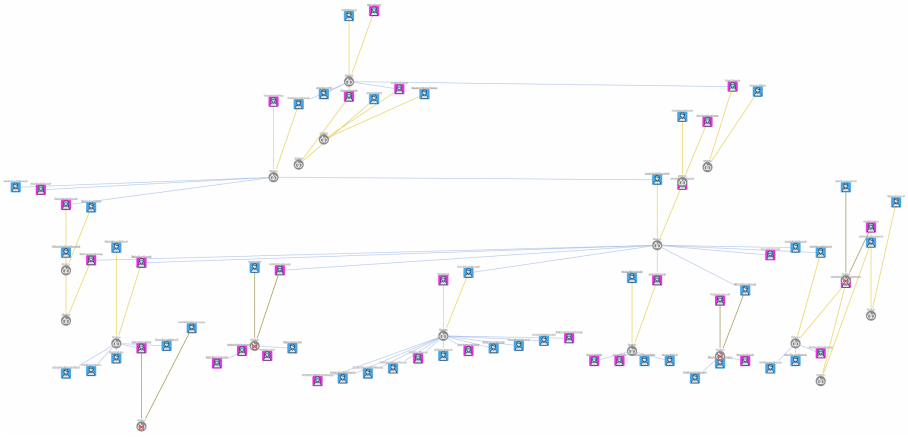
occur is nodes being placed in such a way that their order according to their layers isn't respected (Fig. 6.3b). The old layout also rearranges siblings so that the "by age" layout draws siblings forming a concave shape. Future work might look into how nodes can be arranged to create nicer drawings when arranging nodes by date, e.g. properly center marriage nodes between spouses when there are no associated children.

As seen in the implementation chapter, the new layout algorithm is very general purpose. Some optimizations were done due to knowing the nature of the graphs beforehand, i.e. the layering step, which didn't require a "DAG checking step". However, the algorithm could be further specialized and optimized to fit the semantics of family trees, especially the node reordering step. The implemented solution currently tries to move single nodes around to optimize edge crossings. However, in family tree semantics, it might be beneficial to keep the integrity of families in the drawing by keeping siblings clustered. In this way, the algorithm could try to reorder whole clusters instead of single nodes, which might optimize its running time. Works like the one by Mařík [Mařík, 2016] which creates node orderings that keep siblings clustered could be integrated into the algorithm.

Since this algorithm is implemented in an interactive environment, it needs to run quickly to not keep users waiting for a layout. Due to this, many heuristics and steps in the original work [Gansner et al., 1993] were not implemented as they would increase the computational intensity of the algorithm. For example, the node ordering phase of the implemented solution skips a local optimization step that switches neighboring nodes' positions. And the x-coordinate assignment has

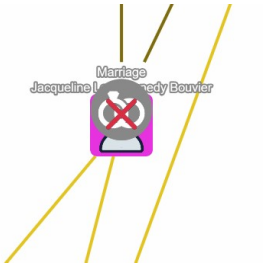


(a) Old "by age" layout.

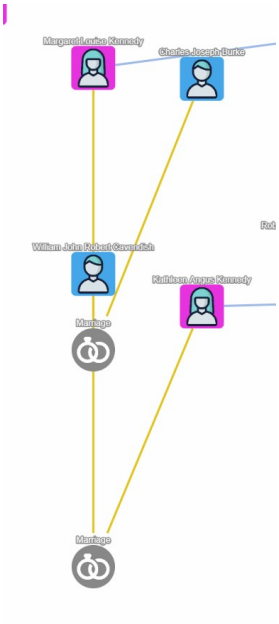


(b) New "by age" layout.

Figure 6.2: Comparison of the old and new layouts in the "by year" variant.



(a) Child overlapping marriage node.



(b) Nodes appearing out of layer order.

Figure 6.3: "By year" layout issues.

File	People count	Marriage count	Node count in old	Node count in new
shakespeare	31	10	31	41
bach	33	8	33	41
elisabeth	40	8	40	48
kennedy family	70	19	70	89
lord of the rings	108	31	108	139

Table 6.1: Family Tree GEDCOM test files information.

many additional heuristics that improve the quality of the drawings which were not implemented. Although they can improve the quality of the results, they are complicated to implement, which would not fit within the time constraints of the dissertation, and fine-tuning would be complicated as the heuristics start to interfere with each other [Gansner et al., 1993]. Gansner's work also presents an alternate method to calculate the coordinates by solving the layering problem in an auxiliary graph [Gansner et al., 1993].

In the context of the coordinate assignment, different methods could also be explored to improve the initial drawing achieved by the initial run of the implemented algorithm. Future work can look into the use of force-directed algorithms to optimize the empty space between nodes on the same layers. Instead of moving freely in space, nodes can be locked to their layer's y-coordinate and let a force-directed algorithm improve their x-coordinate. The forces applied to nodes could also be modeled in a way that follows family tree semantics, e.g. a person node is repelled less by a sibling than by a person from another family. Another possible change to consider is the positioning of marriage nodes. Since marriage nodes and people nodes are placed in distinct layers, the layers with marriage nodes could be placed closer to the parents or the children. This could remedy the increase in graph height to a certain degree. Whether the marriage nodes are positioned closer to the parents or the children would need to be researched.

6.1.1 Performance Comparison

Both the old and new layout algorithms were benchmarked to measure their running times on a set of pre-existing GEDCOM files. Table 6.1 contains details about each file, along with the resultant node counts in each of the graph representations used by each of the layout algorithms. Note that the new layout algorithm has a higher node count since it creates a node for each marriage.

Each file was run three times in each algorithm, and an average of the running time in milliseconds was taken. Figure 6.4 shows the results. From these, it can be observed that the new layout algorithm is slower than the former one. It is not possible, however, to determine if this deficit in performance is proportional, as the differences in running time seem to be constant. More testing with increasingly larger files is needed to determine if this deficit increases with graph size or if it stays the same.

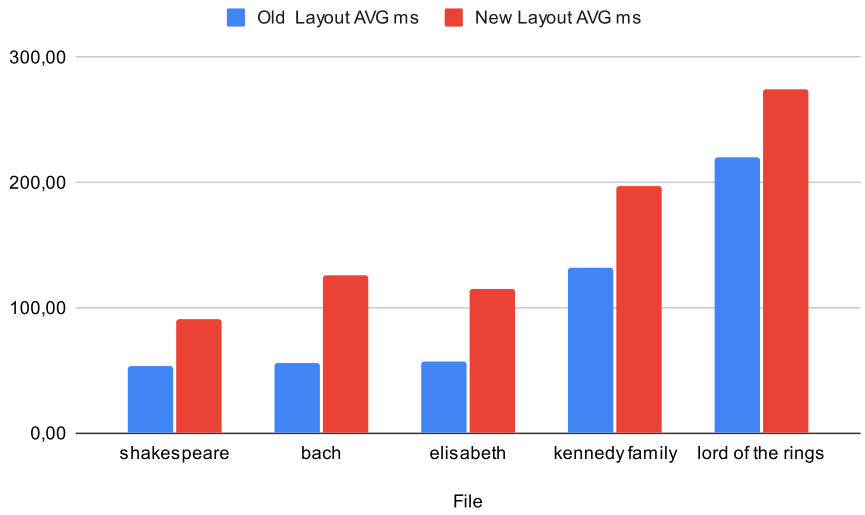


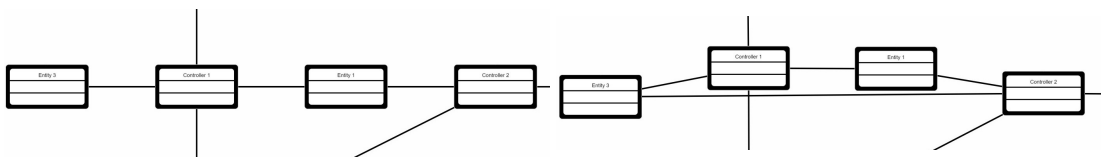
Figure 6.4: Results of the layout algorithm benchmarks.

6.2 UML Class Diagrams

The new algorithm successfully creates drawings of UML class diagrams using the available test data. It improves on the old algorithm by creating a drawing that looks more like a traditional UML Class diagram, with its orthogonal look. The edge routing also prevents cases of edges overlapping nodes, something which often happened in the old layout (e.g. Fig. 6.5). In some cases, edges would overlap each other, giving the impression of a single edge. This is due to the old algorithm being a simpler implementation.

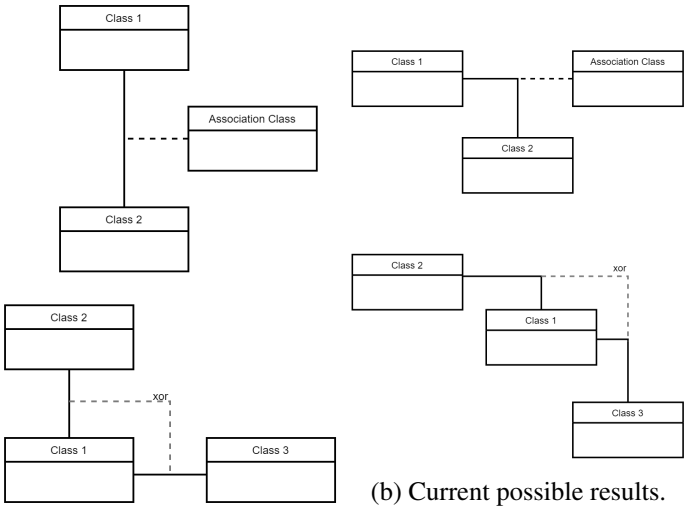
However, the simpler implementation also allowed the algorithm to be more feature complete. The current solution still has some limitations compared with the old one. A situation not handled by the new algorithm is multiple/parallel edges, i.e. groups of edges that connect the same two nodes, as some intermediate stages in the algorithm do not support these situations. The old algorithm handled these cases using Cytoscape.js' renderer, which detects when two edges are parallel and draws them with a slight curve to differentiate them. This issue is better addressed in a later section.

The orthogonalization step can also be extended to take a graph sketch as input, and creates an orthogonal representation that is close to the input sketch [Eiglsperger et al., 2004] [Brandes et al., 2002]. This means the algorithm can compute hints for angles between edges and their



(a) Initial layout. (b) Some nodes slightly moved.

Figure 6.5: Example of edge and node overlap in the former layout.



(a) Desired result.

(b) Current possible results.

Figure 6.6: Desired result of drawing association classes and defined constraints.

shapes. This feature is very useful because it allows the enforcing of certain characteristics in the final drawing, e.g. certain edges can be forced to be completely straight, and directed edges can also be forced to point upwards. It will also be helpful in dealing with association classes and user-defined constraints. These appear as edges connecting other edges and, in regular diagrams, usually "stem off" perpendicularly from the association edges (Fig. 6.6). Currently, the algorithm doesn't ensure this, usually creating 90° bends and the connection points due to there being a node present. However, with the mentioned extension, it could be possible to enforce how these connection points look.

6.2.1 Radial Layout

The update to the radial layout provides satisfactory results. However, it can still be further improved.

Although the expansion tree edges are highlighted, and others are grayed out, some degree of clutter might still happen due to their crossings. Some reordering of nodes within each layer can be done to reduce the number of edges crossing each other or even reduce the length of the edges, effectively reducing the amount of "chaos" in the layout. A possible local optimization can be done between siblings. If there are edges between siblings in the layout, these can be reordered in a way that those edges don't overlap with other siblings. This has the result of clearing making it so these edges aren't obfuscated.

Like in MoireGraphs, there is a possibility to alter an offset to the angular portion of the polar coordinates of all nodes that causes the whole layout to rotate around the root node [Jankun-Kelly and Ma, 2003]. This can be a useful feature in the context of UML class diagrams to apply some orientation to certain kinds of relations. As an example, inheritance relations are usually drawn so that the super-class is located above its sub-classes. The children of the root node can be ordered

in such a way that after applying an angular offset to the layout, all of the root class's sub-classes are drawn below it.

As mentioned in the implementation chapter, when transitioning from one root node to another, the nodes' positions are linearly interpolated over a certain duration. This type of transition frequently features nodes cutting across the center of the layout to reach their destinations. And since all nodes transition to their new positions at the same time, the animation can be quite cluttered. A different approach for this can be taken: instead of linearly interpolating the Cartesian coordinates of the nodes, interpolate the polar coordinates of the nodes, like what is done in MoireGraphs [Jankun-Kelly and Ma, 2003]. This kind of animation will display the layout going through a "rotation" where the nodes assume their new positions. The path the nodes take to get to their new position is a circular one, which fits the radial aesthetic of the graph. There will still be nodes overlapping each other during the transition, but the effect should be less pronounced. The biggest challenge for this would be that node animations in Cytoscape.js are done through an interface that only supports linear interpolation. So an implementation from scratch would be necessary, or an extension could be developed for Cytoscape.js.

Finally, there is a concern about using this layout in graphs with a higher number of nodes. Although layers are sized to fit the dimensions of their nodes, they are not sized to fit all of their nodes. This might result in node overlap when a layer has too many nodes or when a parent doesn't have enough angular spread for its children. The only way for a layer to have more space for its nodes is by increasing its radius, effectively decreasing the angular spread each node occupies in the layer.

6.2.2 Special Case Implementation Ideas

6.2.2.1 Multiple/Parallel Edges

A possible integration of support for multiple/parallel edges is by bundling them into a *bundle edge* and unpacking them at a point where they are no longer an issue. A precaution to take is if one of the edges is directed. In that case, the *bundle edge* should be directed as well and pointing in the direction of that edge.

Where the *bundle edge* should be un-bundled is the main point of investigation. Some ideas are:

1. During the planarization phase after the s-t graph induction but before the directed edge addition.
2. Right after the planarization.
3. After orthogonalization.

With every option, editing of embeddings and the creation of new faces will be necessary. Option 1) is risky since the main authors of the algorithms in that phase assume the non-existence of parallel edges [Bertolazzi et al., 1994] [Eiglsperger and Kaufmann, 2001], the reasons are not

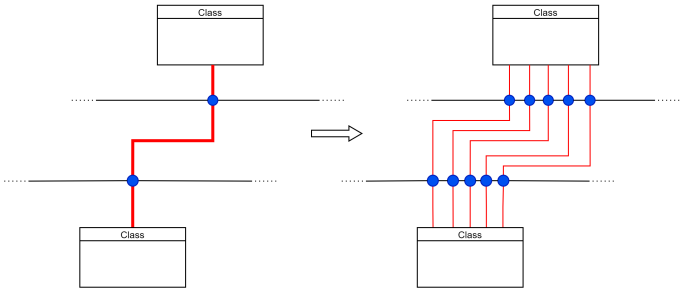


Figure 6.7: Example of the un-bundling of a *bundle edge* with crossings.

specified. Option 2) is past the planarization phase, and the specifications of the algorithms past this point don't make assumptions about parallel edges. A downside of this option is that after planarization, the *bundle edge* might have been crossed by other edges and split with crossing nodes. To un-bundle the edge, these crossing nodes will also need to be multiplied and linked in the correct order (Fig. 6.7). It will also affect the orthogonalization due to the fact that parallel edges usually have the same shape/path. This fact would need to be enforced. Option 3) still has to deal with the multiplication of crossing nodes. However, ensuring all edges have the same shape is easier since they just need to inherit the shape of the *bundle edge*.

6.3 Cytoscape.js Technical Limitations

As mentioned previously, CleanGraph is built on top of Cytoscape.js, which comes pre-packaged with a graph theory model along with a renderer to display graphs. This cuts down on development time as most basic functionality for handling and displaying graphs is already provided. However, some particularities of Cytoscape.js and how CleanGraph works sparked some issues in development.

An example of such issues relates to the UML radial layout and the switching between it and the global layout. In the UML global layout, edges usually have bends within them. These bends are represented by intermediate nodes. This means that, internally, edges with bends are replaced with *segment edges* connected by the intermediate nodes. On the other hand, in the radial layout, edges are straight as the overlap between them isn't a concern. This means that when switching from the global layout to the radial layout, the *segment edges* needs to be removed, and the *original edges* need to be restored. In Cytoscape.js, when an edge is removed, its object isn't deleted from memory. This is so it can be easily restored back into the graph without having to be recreated. So, all that needs to be done in CleanGraph is to keep track of which *segment edges* and *original edges* exist in a collection and restore/remove the appropriate ones at the correct time. The problem starts when dealing with association classes and predefined constraints. These are represented as edges between other edges. However, Cytoscape.js only supports these cases through the use of an extension. This extension creates an auxiliary node at the midpoint of every edge, which allows for edges to be visually connected. For this to happen, the edges involved in these cases must be created through the extension. When one of these edges is deleted, the respective auxiliary

node is removed as well. However, when it is restored, the auxiliary node isn't. This causes the extension to crash to a data inconsistency error which rendered the application unusable. To circumvent this, *original edges* also have to be differentiated between normal ones and ones that were created by the extension. The ones created by the extension were restored by creating a new edge that is identical to the original and forgetting the original one. This differentiation increased the complexity of the implementation of the functionalities and made the application code harder to read, effectively slowing down the implementation process as there were some complex states that needed to be tracked. The solution ended up not being elegant and having something similar to a memory leak. Since deleted edges remain in memory, when switching from the global to the radial layout, the edges created through the extension will be permanently "lost" in memory. If there are multiple switches between the global and radial layouts in a single session, the "restored" edges will be deleted as well, taking up more and more memory until the user refreshes the page.

Chapter 7

Conclusions and Future Work

CleanGraph is an InfoVis system platform developed to address shortcomings in both Family Trees and UML class diagram visualization tools. As a standout, it sports full graph representation, something few other solutions do for Family Trees, and interaction features that provide a productivity boost in certain tasks. However, since the work on the tool was spread out through many fields, the implemented layout algorithms were not optimal, and the interaction with the graphs was quite static.

The purpose of this work is to improve both the layout algorithms and interaction features of CleanGraph. The layout algorithms are expected to improve the efficiency in the use of space while keeping the visualizations readable. The interaction with graphs is expected to rely more on the use of dynamic layouts and animated graph transitions.

The improvement of such aspects relied on an investigation into the state of the art for graph layout and interaction, covering generic graphs, Family Trees, and UML class diagrams. This investigation allowed to gain knowledge about the most popular solutions for graph layout and interaction along with situations in which they are better suited, along with the caveats of using each.

For Family Tree layout, many solutions rely on the visualization of portions of the genealogical graph at a time. This is due to wanting to keep a clean layout while respecting some semantic rules. Although, there is also work on full genealogical graph drawings, which seem like a better fit for CleanGraph.

For UML class diagram layout, full graph representation is strictly necessary, and the studied state of the art reflects that. Hierarchical approaches to drawing class diagrams seem like a popular approach, although Topology-Shape-Metrics approaches seem like a better overall solution for these types of graphs.

On the interaction side, most studied work relies on partial graph representation or graph distortions and providing features to manipulate these visualizations. Of these features, sub-tree collapsing and expanding and root node switching paired up with animated transitions seem like the most interesting. Most of the interactive features for full graph representation are already

implemented in CleanGraph, so improvements to the tool's interaction features will most likely rely on the use of secondary layouts.

After studying the different solutions, decisions were made on which layout algorithms to implement for each of the graph types. For Family Trees, a hierarchical style general purpose layout algorithm was chosen. This meant needing to adapt the graph representation to eliminate the existence of edges between other edges. For UML class diagrams, a Topology-Shape-Metrics algorithm was chosen for its ability to generate orthogonal style drawings. The graph representation for UML class diagrams was also adapted for the same reasons as in Family Trees.

The results of the implementation of the new layout for Family Trees show that the new representation has possible negative effects on the layout. However, it also presents some advantages in the visualization of information. The implemented solution also doesn't produce very good results when using the "by year" layout, with the former implementation having much nicer results.

The new layout algorithm for UML class diagrams provides an improvement over the former one. Edge and node overlap doesn't happen, and the drawings have an orthogonal style which is much closer to traditional UML class diagram drawings. However, it still has some limitations by not being able to handle some special cases, which the former solution was able to handle.

In light of the limitations of the implemented solutions, the following should be considered for future work:

- Family Trees:
  - Specialize the node ordering phase of the Family Trees layout algorithm. Currently, the implementation is a general purpose one and can be specialized to fit better fit Family Tree semantics.
  - Improve the Family Trees coordinates assignment phase. Especially in the "by year" layout, coordinate assignment could be done with this feature in mind to prevent node overlaps. Other methods for coordinate assignment could also be used, like force directed ones which adjust the position of nodes along their layer.
- UML Class Diagrams:
  - Implement support for self associations. These are a common occurrence in UML class diagrams, so it is critical that they are supported by the algorithm.
  - Implement support for multiple associations between the same two classes. This is the same case as self associations.

For the future work to be done in UML class diagrams, the implementation of the extension to the orthogonalization phase to take in a graph sketch can greatly help in the implementation of both features. The implementation of the orthogonalization extension will also help enforce certain characteristics in the drawing and better display UML class diagrams semantics.

The UML class diagrams radial layout can still be further developed to achieve: cleaner drawings through node reordering, better relation representation through angular offsets to the layout, and a cleaner navigation through the change in the node transition style.

Finally, the new layout algorithm and interaction features have not been tested for usability. Analyzing how users interact with the tool to perform certain tasks helps to verify whether the work that was done helped improve the tool as a whole. In UML class diagrams, due to limitations on the XMI parser that generates the graph information for the layouts, it was not possible to do these tests. The efforts taken to try to fix these limitations caused time constraints that ended up making it unfeasible to test the other features.

Bibliography

[Battista et al., 1998] Battista, G. D., Eades, P., Tamassia, R., and Tollis, I. G. (1998). *Graph drawing: algorithms for the visualization of graphs*. Prentice Hall PTR. ISBN: 978-0-13-301615-4.

[Bertolazzi et al., 1994] Bertolazzi, P., Battista, G. D., Liotta, G., and Mannino, C. (1994). Upward drawings of triconnected digraphs. *Algorithmica*, 12(6):476–497. DOI: 10.1007/BF01188716.

[Booch et al., 1999] Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide*, volume 10. Addison Wesley, 2 edition. ISBN: 0-201-57168-4.

[Brandes et al., 2002] Brandes, U., Eiglsperger, M., Kaufmann, M., and Wagner, D. (2002). Sketch-driven orthogonal graph drawing. In Goodrich, M. T. and Kobourov, S. G., editors, *Graph Drawing*, pages 1–11, Berlin, Heidelberg. Springer Berlin Heidelberg. DOI: 10.1007/3-540-36151-0_1.

[Collins English Dictionary, 2021] Collins English Dictionary (2021). Family tree definition and meaning. https://www.collinsdictionary.com/dictionary/english/family-tree. Accessed: 2021-03-01.

[Cytoscape.js, 2021] Cytoscape.js (2021). Using layouts. https://blog.js.cytoscape.org/2020/05/11/layouts/#force-directed-layouts. Accessed: 2022-03-01.

[Eades, 1984] Eades, P. (1984). A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160.

[Eichelberger, 2002] Eichelberger, H. (2002). Aesthetics of class diagrams. In *Proceedings of First International Workshop on Visualizing Software for Understanding and Analysis, 26-26 June 2002, Paris, France*, page 23–31. DOI: 10.1109/VISSOF.2002.1019791.

[Eichelberger, 2006] Eichelberger, H. (2006). On Class Diagrams, Crossings and Metrics. In Jünger, M., Kobourov, S., and Mutzel, P., editors, *Graph Drawing*, volume 5191 of *Dagstuhl Seminar Proceedings (DagSemProc)*, pages 1–12, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. DOI: 10.4230/DagSemProc.05191.5.

[Eiglsperger et al., 2004] Eiglsperger, M., Gutwenger, C., Kaufmann, M., Kupke, J., Jünger, M., Leipert, S., Klein, K., Mutzel, P., and Siebenhaller, M. (2004). Automatic layout of uml class diagrams in orthogonal style. *Information Visualization*, 3(3):189–208. DOI = 10.1057/palgrave.ivs.9500078.

[Eiglsperger and Kaufmann, 2001] Eiglsperger, M. and Kaufmann, M. (2001). An approach for mixed upward planarization. In Dehne, F., Sack, J.-R., and Tamassia, R., editors, *Algorithms and Data Structures, WADS 2001*, volume 2125 of *Lecture Notes in Computer Science*, page 352–364, Berlin, Heidelberg. Springer. ISBN: 978-3-540-44634-7, DOI: 10.1007/3-540-44634-6_33.

[Eiglsperger and Kaufmann, 2002] Eiglsperger, M. and Kaufmann, M. (2002). Fast compaction for orthogonal drawings with vertices of prescribed size. In Mutzel, P., Jünger, M., and Leipert, S., editors, *Graph Drawing, GD 2001*, volume 2265 of *Lecture Notes in Computer Science*, page 124–138, Berlin, Heidelberg. Springer. DOI: 10.1007/3-540-45848-4_11.

[Eiglsperger et al., 2003] Eiglsperger, M., Kaufmann, M., and Siebenhaller, M. (2003). A topology-shape-metrics approach for the automatic layout of uml class diagrams. In *Proceedings of the 2003 ACM Symposium on Software Visualization (SoftVis '03), 2003, New York, USA*, SoftVis '03, page 189–ff, New York, USA. Association for Computing Machinery. DOI: 10.1145/774833.774860.

[Franz et al., 2016] Franz, M., Lopes, C. T., Huck, G., Dong, Y., Sumer, O., and Bader, G. D. (2016). Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics*, 32(2):309–311. DOI: 10.1093/bioinformatics/btv557.

[Friendly and Wainer, 2021] Friendly, M. and Wainer, H. (2021). *A History of Data Visualization and Graphic Communication*. Harvard University Press. ISBN: 978-0-674-25903-4.

[Fruchterman and Reingold, 1991] Fruchterman, T. M. J. and Reingold, E. M. (1991). Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164. DOI: 10.1002/spe.4380211102.

[Fößmeier and Kaufmann, 1996] Fößmeier, U. and Kaufmann, M. (1996). Drawing high degree graphs with low bend numbers. In Brandenburg, F. J., editor, *Graph Drawing, GD 1995*, volume 1027 of *Lecture Notes in Computer Science*, page 254–266, Berlin, Heidelberg. Springer. DOI: 10.1007/BFb0021809.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley. ISBN: 978-0-201-63361-0.

[Gansner et al., 1993] Gansner, E., Koutsofios, E., North, S., and Vo, K.-P. (1993). A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230. DOI: 10.1109/32.221135.

[Gansner et al., 2015] Gansner, E. R., Koutsofios, E., and North, S. C. (2015). Drawing graphs with dot. URL: https://graphviz.org/pdf/dotguide.pdf, Accessed: 2022-07-30.

[Goldschmidt and Takvorian, 1994] Goldschmidt, O. and Takvorian, A. (1994). An efficient graph planarization two-phase heuristic. *Networks*, 24:69–73. DOI: 10.1002/net.3230240203.

[Herman et al., 2000] Herman, I., Melancon, G., and Marshall, M. (2000). Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43. DOI: 10.1109/2945.841119.

[Hu et al., 2012] Hu, H., Fang, J., Lu, Z., Zhao, F., and Qin, Z. (2012). Rank-directed layout of uml class diagrams. In *Proceedings of the First International Workshop on Software Mining, August 2012*, SoftwareMining '12, page 25–31, New York, USA. Association for Computing Machinery. DOI: 10.1145/2384416.2384420.

[Jankun-Kelly and Ma, 2003] Jankun-Kelly, T. and Ma, K.-L. (2003). Moiregraphs: radial focus+context visualization and interaction for graphs with visual nodes. In *Procedings of the IEEE Symposium on Information Visualization 2003 (IEEE Cat. No.03TH8714), Seattle, USA*, page 59–66. DOI: 10.1109/INFVIS.2003.1249009.

[Keller et al., 2010] Keller, K., Reddy, P., and Sachdeva, S. (2010). Family tree visualization. *Course Project Report*. URL: http://vis.berkeley.edu/courses/cs294-10-sp10/wiki/images/f/f2/Family_Tree_Visualization_-_Final_Paper.pdf, Accessed: 2022-07-30.

[Kobourov, 2012] Kobourov, S. G. (2012). Spring Embedders and Force Directed Graph Drawing Algorithms. *arXiv e-prints*, page arXiv:1201.3011.

[Locoro et al., 2017] Locoro, A., Cabitza, F., Actis-Grosso, R., and Batini, C. (2017). Static and interactive infographics in daily tasks: A value-in-use and quality of interaction user study. *Computers in Human Behavior*, 71:240–257. DOI: 10.1016/j.chb.2017.01.032.

[Martins, 2021] Martins, A. A. d. A. (2021). Cleangraph - graph viewing and editing for family trees and uml class diagrams. Master's thesis, Universidade do Porto. URL: https://hdl.handle.net/10216/135507.

[Mařík, 2016] Mařík, R. (2016). On large genealogical graph layouts. In *Proceedings of the 16th ITAT Conference Information Technologies - Applications and Theory, September 15-19, 2016, Tatranské Matliare, Slovakia*, volume 1649, page 218–225.

[McGuffin and Balakrishnan, 2005] McGuffin, M. and Balakrishnan, R. (2005). Interactive visualization of genealogical graphs. In *Proceedings of the IEEE Symposium on Information Visualization, INFOVIS 2005, 23-25 Oct. 2005, Minneapolis, USA*, page 16–23. DOI: 10.1109/INFVIS.2005.1532124.

[Nikolov, 2016] Nikolov, N. (2016). Sugiyama algorithm. In *Encyclopedia of Algorithms*, page 2162–2166. Springer. DOI: 10.1007/978-1-4939-2864-4_649.

[OMG, 2017] OMG (2017). About the unified modeling language specification version 2.5.1. https://www.omg.org/spec/UML. Accessed: 2022-07-04.

[Purchase, 1997] Purchase, H. (1997). Which aesthetic has the greatest effect on human understanding? In DiBattista, G., editor, *Graph Drawing, 5th International Symposium, GD '97, September 18–20, 1997, Rome, Italy*, Lecture Notes in Computer Science, page 248–261. Springer. DOI: 10.1007/3-540-63938-1_67.

[Purchase, 2014] Purchase, H. C. (2014). Twelve years of diagrams research. *Journal of Visual Languages & Computing*, 25(2):57–75. DOI: 10.1016/j.jvlc.2013.11.004.

[Purchase et al., 2000] Purchase, H. C., Allder, J.-A., and Carrington, D. (2000). User preference of graph layout aesthetics: A uml study. In Marks, J., editor, *Graph Drawing, 8th International Symposium, GD 2000, September 2000, Colonial Williamsburg, USA*, Lecture Notes in Computer Science, page 5–18. Springer. DOI: 10.1007/3-540-44541-2_2.

[Purchase et al., 1997] Purchase, H. C., Cohen, R. F., and James, M. I. (1997). An experimental study of the basis for graph drawing algorithms. *ACM Journal of Experimental Algorithmics*, 2:4–es. DOI: 10.1145/264216.264222.

[Purchase et al., 2001] Purchase, H. C., Mcgill, M., Colpoys, L., and Carrington, D. (2001). Graph drawing aesthetics and the comprehension of uml class diagrams: an empirical study. In *Proceedings of the Australian Symposium on Information Visualization, InVis', 2001, Australia*. ISBN: 0909925879.

[Pálsson, 2002] Pálsson, G. (2002). The life of family trees and the book of icelanders. *Medical Anthropology*, 21(3–4):337–367. DOI: 10.1080/01459740214078.

[Riaz and Ali, 2011] Riaz, F. and Ali, K. M. (2011). Applications of graph theory in computer science. In *Proceedings of the 2011 Third International Conference on Computational Intelligence, Communication Systems and Networks, CICSYN, 26-28 July 2011, Bali, Indonesia*, page 142–145. DOI: 10.1109/CICSyN.2011.40.

[Seemann, 1997] Seemann, J. (1997). Extending the sugiyama algorithm for drawing uml class diagrams: Towards automatic layout of object-oriented software diagrams. In *Graph Drawing, 5th International Symposium, GD '97, September 18–20, 1997, Rome, Italy*, GD '97, page 415–424, Berlin, Heidelberg. Springer-Verlag. DOI: 10.1007/3-540-63938-1_86.

[Sharpe, 2011] Sharpe, M. (2011). *Family Matters: A History of Genealogy*. Casemate Publishers. Google-Books-ID: qgXMDwAAQBAJ, ISBN: 978-1-84468-650-6.

[Smith, 1925] Smith, W. H. (1925). Graphic statistics in management. *Nature*, 115. DOI: 10.1038/115454d0, ISBN: 10.1038/115454d0.

[Sorapure, 2019] Sorapure, M. (2019). Text, image, data, interaction: Understanding information visualization. *Computers and Composition*, 54:102519. DOI: 10.1016/j.compcom.2019.102519.

[Sugiyama et al., 1981] Sugiyama, K., Tagawa, S., and Toda, M. (1981). Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125. DOI: 10.1109/TSMC.1981.4308636.

[Tamassia, 1987] Tamassia, R. (1987). On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16(3):421–444. DOI: 10.1137/0216030.

[Victorelli et al., 2020] Victorelli, E. Z., Dos Reis, J. C., Hornung, H., and Prado, A. B. (2020). Understanding human-data interaction: Literature review and recommendations for design. *International Journal of Human-Computer Studies*, 134:13–32. DOI: 10.1016/j.ijhcs.2019.09.004.

[Visual Paradigm team, 2021] Visual Paradigm team (2021). What is unified modeling language (uml)? https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/. Accessed: 2022-02-06.