Masaryk University
Faculty of Informatics

A library for drawing directed graphs

Bachelor's Thesis

Miroslav Demek

Brno, Spring 2020

Masaryk University
Faculty of Informatics

A library for drawing directed graphs

Bachelor's Thesis

Miroslav Demek

Brno, Spring 2020

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Miroslav Demek

Advisor: RNDr. Petr Ročkai, Ph.D.

Acknowledgements

I want to express my deepest gratitude to my thesis supervisor RNDr. Petr Ročkai, Ph.D. for his feedback, patience with my questions, and always finding time for me. I would also like to thank all of my friends and loved ones for their support and encouragement. Lastly, my special thanks go to my hands for being clumsy and spilling water on my laptop, forcing me to use the old one, thereby removing many distractions and helping me to focus on my thesis.

Abstract

The goal of this thesis is to implement a C++ library for creating visualizations of directed graphs. The library allows the user to create a graph by adding vertices and edges and create a hierarchical layout of this graph in space which is described by the positions of vertices and control points of edges. The thesis first gives an introduction to graph drawing, then the implemented algorithms are described in detail and the interface and usage of the library are illustrated on a sample application. Lastly, the quality of drawings produced by the library is evaluated by comparing them with drawings produced by a popular graph drawing tool *dot*.

Keywords

graph, graph drawing, layered drawings, hierarchical drawings, Sugiyama framework, C++ library

Contents

Introduction

Data can often be modeled as a graph, where the individual entities are represented as nodes and their relationships as edges. This structure can then be visualized through the process of graph drawing. Visualizing data in this way has several benefits, such as making them easier to understand and to grasp their overall structure or to gain new insights.

There are many different applications for graph drawing. For example, visualizing data structures, computer networks, call graphs, UML (Unified Modeling Language) diagrams or entity-relationship diagrams. However, the applications are not restricted to computer science and arise in other scientific areas as well.

Graph drawing uses many different drawing conventions, each suitable for different input graphs and different applications. This thesis focuses on hierarchical drawings of directed graphs, where nodes are drawn on horizontal lines such that the edges flow in a common direction. One of the most popular tools for producing these is *dot*, which is a part of the *Graphviz*[1] software suite. It takes a file with a simple textual description of the input graph and produces its visualization in one of the supported formats.

The aim of this thesis is to implement a simple C++ library which provides a similar functionality but eliminates the need to generate a file and invoke a separate program. Instead, it provides an API (Application Programming Interface) for incrementally describing the input graph and producing a description of its layout in space. This description is independent of a particular output device and consists only of 2D coordinates of the nodes and control points of edges.

In Chapter 1, I describe the basic concepts and approaches to graph drawing. Chapter 2 contains a detailed description of the graph drawings algorithms I implemented for my library. In Chapter 3, I describe the interface of the library and show its usage on a sample application for producing SVG images of graphs. Finally, in Chapter 4, I compare the quality and readability of images produced by my library with images produced by *dot*.

---

1. `https://graphviz.org/`

1 Graph drawing

This chapter describes basic graph drawing concepts. First, I briefly introduce different drawing conventions and aesthetic criteria for readable drawings of graphs, and then go over two main classes of layout algorithms. The main focus are algorithms for producing hierarchical drawings of directed graphs since these are the subject of this thesis.

1.1 Preliminaries

A graph is a mathematical structure which represents a set of objects where some of the objects are in pairwise relationships. Formally, a graph is a pair $G = (V, E)$ consisting of a set of vertices (nodes) V and a set of edges E. The vertices represent the objects and an edge represents a relationship between two of those objects, thus an edge is a set of two vertices $\{u, v\}$.

In a directed graph the edges are ordered pairs of vertices (u, v), and represent only a one-directional relationship from u to v but not the other way around.

1.2 Drawing conventions

The goal of graph drawing is to produce a graphical representation of a graph. Usually, vertices of the graph are depicted as shapes, such as rectangles or circles, placed in the 2D plane. The edges are generally mapped to poly-lines, thus such drawings are called *poly-line drawings*. Alternatively, the line segments can be replaced by a smooth curve. There are two special cases of poly-line drawings: *straight-line drawings*, where each edge is represented as a single line, and *orthogonal drawings*, where the edges are formed as a series of alternating horizontal and vertical segments. A drawing where no two edges intersect is called a *planar drawing*.

In *layered* (*hierarchical*) *drawings* the vertices are split into multiple layers in such a way that the edges aim in a common direction (i.e. left to right or top to bottom). The vertices on the same layer are constrained to lay on a single line and the layers are drawn in parallel as

either vertical or horizontal lines. In this thesis I consider the layers to be horizontal and the edges to be aimed from top to bottom. Alternatively, in *radial drawings* the layers are drawn as concentric circles with edges pointing outward.

Layered drawings are especially suitable for representing hierarchies since they clearly convey the information of certain vertices being "superior" to others by putting them on a higher layer. Directed graphs are hierarchical in nature since an edge (u, v) can be interpreted as u preceding v in the hierarchy and thus are often visualized using layered drawings.

Yet another way to draw a graph is a *circular layout*, where the vertices are split into clusters, and the vertices in the same cluster are drawn along the circumference of a circle.

1.3 Aesthetic criteria

For a given graph, there are many possible representations. Usually, we want to find one which clearly conveys the information contained in the graph. To obtain such a drawing, we first need to establish what a readable drawing looks like.

That can be achieved by introducing a set of aesthetic criteria, which are usually expressed as optimization goals that can be followed by specific algorithms. The following are the most commonly used criteria [1].

- **Number of edge crossings**: The number of edge crossings should be minimized. This is one of the most important factors influencing the quality of the drawing [2]. A high number of edge crossings means it is hard to follow edges and to determine which vertices are connected and which are not.
- **Bends in edges**: For similar reasons, it is preferable to minimize the number of bends in edges. That is especially important in orthogonal layouts.
- **Edge length**: The total length of edges should be minimized.
- **Symmetries**: If a graph contains symmetrical information, it should be displayed in its layout.

- **Angle maximization**: The minimum angle between edges leaving a vertex should be maximized and as uniform across the whole drawing as possible.

Specifically for directed graphs, there is often one additional criterion [3].

- **Uniform flow**: Edges should be aimed in the same direction. That makes it easier to follow directed paths and to determine the source and sink vertices.

It is important to note that it is impossible to optimize all of the above criteria at the same time because they often contradict each other. For example, keeping edges short might result in more crossings. Another issue is that some of these problems, such as crossing reduction or finding subgraphs with a high degree of symmetry, are computationally hard, which means we need to rely on various simplifications and heuristics.

1.4 Force-directed algorithms

This class of algorithms is mainly used for producing straight-line drawings of general graphs, without prior knowledge of any of their structural properties. However, the force-directed methods can be adapted to take into account the direction of edges and they can be used for directed graphs as well [4].

The force-directed algorithms work by simulating a physical model with repulsive and attractive forces between vertices until an equilibrium point is reached. This system of vertices represented as physical objects is described by an objective function corresponding to the energy of the system. The value of this function is minimized when the distance between adjacent vertices is near some ideal value, and non-adjacent vertices are well spaced. Thus, the final layout of the graph is obtained by finding a (often only local) minimum of this function.

One of the first force-directed algorithms was introduced by Eades [5]. It works by connecting adjacent vertices with springs and repelling each pair of non-adjacent vertices.

Later, Fruchterman and Reingold [6] refined this method by adding a notion of temperature which limits the displacement of a vertex in each iteration to a specific maximum value. The temperature gradually decreases and in later stages when the layout is closer to optimal, the adjustments are finer and finer.

Another algorithm introduced by Kamada and Kawai [7] defines the ideal distance between vertices as the graph-theoretic distance between them (the length of the shortest path) and the vertices repel/attract each other if they are closer/further apart then this distance.

1.5 Sugiyama framework

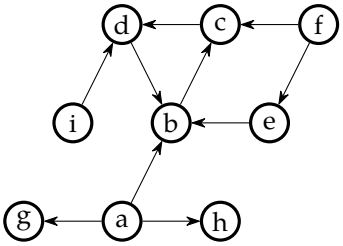
Although force-directed algorithms can be used for drawing directed graphs, these are usually drawn using algorithms for layered layouts. Most algorithms for producing layered drawings of directed graphs follow the *Sugiyama framework* (*Sugiyama method*) [8].

It splits the problem of creating a layout into multiple steps, each with its own algorithms. Since in layered drawings the edges should aim in a common direction, the first step is to remove all directed cycles by reversing suitable edges. That enables us to assign the vertices to layers such that each edge goes from a higher layer to a lower one. As a postprocessing phase of the layer assignment edges which span multiple layers are broken down by using dummy vertices. Then the vertices on each layer are reordered to reduce the number of crossings. Lastly, the final coordinates of vertices are calculated. The individual steps are illustrated in Fig. 1.1.

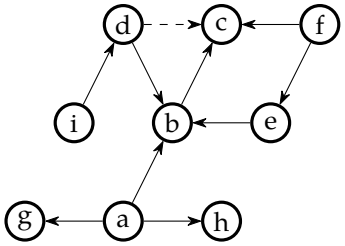
1.5.1 Cycle removal

First, we need to make the input graph acyclic, otherwise it would be impossible to aim all edges in one direction. This is usually done by reversing some subset of edges and then restoring them back to their original orientation in the final drawing.

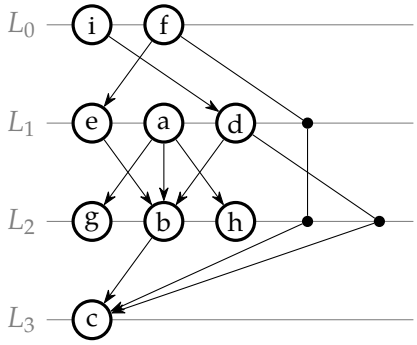
One of the simplest solutions is to take any linear ordering of the vertices, for example, the one generated by DFS (depth first search) and reverse all edges going from a vertex with a higher order to a



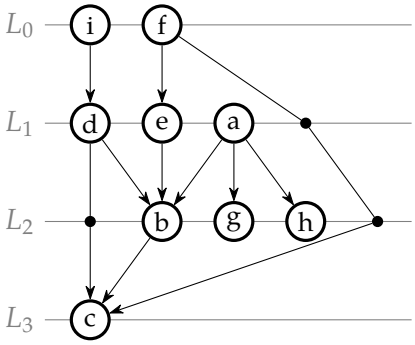
(**a**) Original graph



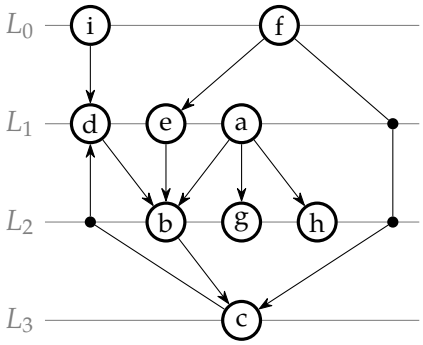
(**b**) After cycle removal (reversed edges are dashed)



(**c**) After layer assignment (dots represent dummy vertices)



(**d**) After crossing reduction



(**e**) After coordinate assignment and restoring edge direction

Figure 1.1: Steps of the Sugiyama framework (modified from [9])

vertex with a lower one [9]. However, this approach doesn't give any guarantees on the number of edges that need to be reversed.

Instead, it might be desirable to find the minimum set of edges whose reversal makes the graph acyclic since these edges go against the final flow of the drawing and impair its readability. This is the subject of the minimum feedback arc set problem, which is unfortunately NP-hard [10]. Equivalently, it can also be stated as its complimentary problem: the maximum acyclic subgraph problem.

A greedy heuristic for solving this problem was proposed by Berger and Shor [11]. It guarantees that the number of reversed edges is at most $|E|/2$. Later, the algorithm was refined by Eades et al. [12] who improved the bound to at most $|E|/2 - |V|/6$ edges.

1.5.2 Layer assignment

During this step, each vertex is assigned to a layer such that every edge goes from a higher layer to a lower one. Constructing this layering is only possible for acyclic graphs, which is the reason why all cycles need to be removed in the cycle removal phase.

Furthermore, the crossing reduction step requires the resulting layering to be proper, which means each edge should connect only adjacent layers. If the layering is not proper it can be transformed into one by splitting edges which span across multiple layers into several sub-edges using newly added dummy vertices.

The number of dummy vertices negatively impacts the performance of subsequent phases, and they cause edges to be longer and have more bends, which is undesirable. Thus, a common requirement is to find a layering with the minimum number of dummy vertices. This can be achieved for example by linear programming or using the network simplex algorithm by Gansner et al. [3].

Two important parameters of the layering are the width (the maximum number of vertices in a layer) and the height (the number of layers). Finding a layering with the minimum height given an upper bound on the width is equivalent to the precedence-constrained multiprocessor scheduling (PCMS) problem [9], which is NP-hard [10]. The subject of the problem is to assign n tasks to m processors such that the precedence constraints between tasks are satisfied and the total computation time is minimized.

We can reduce the layering problem to PCMS problem in the following way. Each vertex corresponds to one task and all tasks take one unit of time to execute. An edge (u, v) indicates that the task u has to be executed before v. Finally, the maximum width of the layering is the number of available processors. Then, if we obtain the optimum scheduling, we can assign the tasks run at time i to layer i, which results in a correct layering with the minimum height and the width less then the given bound. Thus, to find a layering we can use any algorithm for solving the PCMS problem. One such popular algorithm is the Coffman-Graham algorithm [13].

A layering with the minimum height can also be constructed using the longest-path algorithm [9]. However, it performs poorly in terms of the number of dummy vertices and the resulting layering tends to be wide, especially at the bottom.

1.5.3 Crossing reduction

The goal of this phase is to minimize the total number of edge crossings. Given a layering, the number of crossings depends only on the relative order of vertices on their layers and not on their final positions. Thus, to minimize crossings, we only need to find a suitable permutation of each layer. However, the problem of finding permutations of only two layers which minimize crossings between them is NP-hard [14] and even if we fix the ordering of one of the layers, it remains NP-hard [15]. In the following discussion it is assumed that the input is a graph with a proper layering.

In practice, this problem is usually solved by a layer-by-layer sweep [9]. First, an initial ordering is chosen. Then we go through the layers from top to bottom and during each iteration we hold the previous layer fixed and reorder the current layer. Then we go through the same process again, but this time going from the bottom layers up. This process is repeated until no further improvements can be made or until the changes in the number of crossings get sufficiently small.

The main problem in this process is to reduce crossings between two layers with the ordering of vertices in one layer being fixed: the one-sided crossing minimization problem. Two main heuristics for solving this problem are the barycenter [8] and median heuristic [15]. Both of them work by assigning a weight to each vertex in the

free layer and then sorting it based on this weight. In the barycenter heuristic the weight of a vertex is the average position of its neighbors in the fixed layer, and in the median heuristic it is the median position. An advantage of the median heuristic is that it guarantees that the number of crossings is at most $3 \cdot OPT$ [15]. No such bound is known for the barycenter heuristic.

1.5.4 Coordinate assignment

During this step, the final positions of vertices are calculated and dummy vertices are replaced with bends in edges. The assignment must preserve the ordering of vertices determined in the previous step and keep a certain minimum distance between vertices and between layers. Determining the y-coordinates is trivial. Thus, the main problem is computing the x-coordinates. To obtain an aesthetically pleasing drawing, the coordinates should be calculated in such a way that the edges have as few bends as possible and each vertex is positioned close to its neighbors.

One way to solve the problem is quadratic programming [8] which uses an objective function that combines two goals: closeness of connected vertices and centering a vertex between its neighbors.

Gansner et al. [3] propose two different solutions. One is a heuristic solution which uses a similar approach as the layer-by-layer sweep for crossing reduction. It sweeps up and down over the layers and at each iteration applies a set of heuristics for improving different aspects of the assignment. The other solution works by applying their network simplex algorithm for layer assignment to a modified version of the input graph.

Another alternative is an algorithm by Brandes and Köpf [16]. They construct four "extreme" layouts with four symmetrical biases and then combine them into the final layout [16].

2 Implemented algorithms

In this chapter, I present the drawing algorithm I implemented for the library. It is an implementation of Sugiyama framework since that is the most common way to create layouts of directed graphs. In each section, I go over a single step of the framework and describe the algorithm I chose for the implementation in detail. I mostly followed the implementation of *dot*, which is based on the paper by Gansner et al. [3].

The input to the algorithm is a graph without multi-edges but possibly containing loops and not necessarily connected. The output is a set of coordinates of nodes and control points of edges. The nodes are represented as circles and the edges as poly-lines. In addition to the graph, the input consists of several attributes governing the size and spacing of elements in the drawing. These include the size of nodes (their radius), the minimum distance between two nodes and between two layers and two parameters describing the size of loops (see 2.6). The sizes of nodes in the drawing are uniform and the given size applies to all of them.

The Sugiyama framework is designed for connected graphs, but the input graph might be disconnected, thus it first needs to be split into its connected components. For this purpose I use depth first search. Then the Sugiyama framework can be applied to each component separately. In the final layout, I combine the layouts of the individual components simply by placing them in a row next to each other.

2.1 Definitions and notation

The purpose of this section is to describe the notation and terminology I use in the following sections. A directed graph $G = (V, E)$ is a pair consisting of a set of vertices V and a set of edges $E \subseteq V \times V$. An edge $e = (u, v)$ is an ordered pair of vertices. By $N^+(u) = \{v \mid (u, v) \in E\}$ I denote the set of successors of u and by $N^-(u) = \{v \mid (v, u) \in E\}$ the set of predecessors of u. Similarly, $d^+(u)$ is the out-degree and $d^-(u)$ the in-degree of u.

A *layering* (ranking) $\mathcal{L} = (L_0, L_1, \ldots, L_{h-1})$ of G is the partitioning of V into layers L_i such that if $(u, v) \in E$ and $u \in L_i$ and $v \in L_j$ then

$i < j$. The height of $\mathcal{L}$ is h. If $u \in L_i$, then $l(u) = i$ is the *layer* (rank) of u. A *hierarchy* is a graph with a layering.

The vertices on each layer are ordered and the i-th vertex on layer L is denoted $L[i]$. The position of u on its layer is denoted $pos(u)$, the vertex immediately before it $prev(u)$ and the vertex immediately after it $next(u)$.

I define the *span* of an edge (u, v) as $l(v) - l(u)$ and the *slack* as $span(u, v) - 1$. *Tight edges* have span 1 (i.e. they connect adjacent layers). *Long edges* cross multiple layers and have span greater then 1.

A *proper layering* is a layering where all edges are tight. If a layering is not proper it can be transformed into one by subdividing each long edge (u, v) with span s into edges (u, d_1), (d_1, d_2), $\ldots$, (d_{s-2}, d_{s-1}), (d_{s-1}, v) where d_i are new vertices called *dummy vertices* and $l(d_i) = l(u) + i$. Each dummy vertex u has $d^+(u) = d^-(u) = 1$. The edges in a proper layering are also called *segments* and a segment between two dummy vertices is called an *inner segment*.

2.2 DFS cycle removal

First, all the cycles need to be removed. Otherwise, it would be impossible to construct a layering. For this purpose I use a DFS based approach. The main reasons are that it is both fast and easy to implement.

The idea of the algorithm is to repeatedly pick an unvisited vertex and perform a depth first search of the graph starting with this vertex until the search has visited all vertices. Using DFS, we can mark each vertex with the time we finished its processing. If we sort the vertices by this mark in decreasing order and reverse all edges which go from right to left (from a vertex with a lower mark to a vertex with a higher one) we obtain an acyclic graph.

Special treatment needs to be given to two-cycles. These are cycles formed by two edges (u, v) and (v, u). By reversing one of them we would obtain a multi-edge. However, all subsequent steps assume that the graph does not contain any multi-edges. Thus the edge is removed instead of reversed.

Another special case are loops which need to be removed as well. To make sure there is enough room to reinsert the loop at the end of

Algorithm 1: DFS cycle removal

```
Procedure DFS_reverse(u) begin
    u.on_stack ← true
    foreach v ∈ N⁺(u) do
        if v.on_stack then
            if (v, u) ∈ E or v = u then
                remove edge (u, v)
            else
                reverse edge (u, v)
        else if not v.done then
            DFS_reverse(v)
    v.done ← true
    u.on_stack ← false

// initialization
foreach u ∈ V do
    u.on_stack ← false
    u.done ← false
foreach u ∈ V do
    if not v.done then DFS_reverse(u)
```

the drawing algorithm, we temporarily increase the size of the vertex with the loop to reserve space for it.

The cycle removal algorithm is described in Alg. 1. Each vertex has two marks: `on_stack`, meaning a vertex is being processed and `done`, meaning a vertex has already been finished. Both are initialized to *false*. We go through all vertices and start the search from each unfinished vertex. When the search starts processing a vertex it is marked as `on_stack` and then it iterates through all of its successors. If a successor is `on_stack` we know it is still being processed, and thus its processing will end later then the current vertex and it would have a higher time mark, so the edge connecting them needs to be reversed or removed if its a two-cycle or a loop. Otherwise, if the successor hasn't been processed yet, we can recursively process it. When finishing a vertex we need to remove the `on_stack` mark and mark it `done`.

2.3 Layer assignment

The goal of this phase is to produce a proper layering for the input graph, i.e. assign each vertex to a layer such that each edge goes from a higher layer to a lower one. For the purposes of this section, edge length is defined as the span of the edge.

For the implementation I used the network simplex algorithm [3], which is an adaptation of the network simplex algorithm used for solving the minimum cost flow problem. It is designed to find a layering with the minimum total edge length or equivalently with the minimum number of dummy vertices. This is desirable since low total edge length is one of the criteria for readable drawings, but also because the running time of the subsequent steps depends on the number of dummy nodes.

Formally, the problem of finding the layering with the minimum edge length can be stated as the following linear program:

$$\begin{aligned} &\text{minimize} \sum_{(u,v)\in E} l(v) - l(u) \\ &\text{subject to: } l(v) - l(u) > 0 \quad \forall (u,v) \in E \end{aligned}$$

2.3.1 Spanning tree solutions

The main idea of the algorithm is to keep a spanning tree of the graph and repeatedly choose a suitable tree edge and replace it by a non-tree edge obtaining a new spanning tree and decreasing the total edge length in the process.

Each spanning tree of the input graph corresponds to a set of equivalent solutions to the layer assignment problem. One of the solutions can be obtained in the following way. First, we pick a vertex and assign it to an arbitrary layer. Then we repeatedly find an unranked vertex u adjacent in the spanning tree to a ranked vertex v and assign it to the preceding (if $(u,v) \in E$) or the following (if $(v,u) \in E$) layer. An important property of such a solution is that all edges in the spanning tree are tight. This property needs to be preserved by all transformations on the spanning tree.

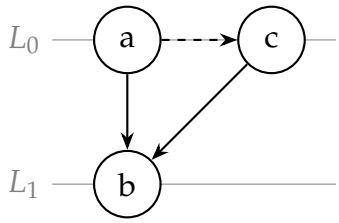


Figure 2.1: Infeasible spanning tree of a graph (non-tree edges are dashed)

A spanning tree is feasible if it induces a feasible layering, i.e. layering satisfying the constraint $l(v) - l(u) > 0, \forall (u, v) \in E$. Not every spanning tree is feasible. In the example in Fig. 2.1 the edges (a, b) and (c, b) form a spanning tree which is infeasible, since the induced layering would force (a, c) to connect nodes on the same layer.

Algorithm 2: Obtaining initial feasible tree (from [3])

```
Procedure feasible_tree(G) begin
    initialize_layering(G)
    tree ← initialize_tree(G, root)
    while tree.size < |V| do
        e ← non-tree edge incident on the tree with minimum slack
        d ← slack(e)
        if (end node of e) ∈ tree then
            d ← −d
        foreach u ∈ tree do
            l(u) ← l(u) + d
        add e to tree
    return tree
```

The first step of the network simplex algorithm is to find an initial feasible tree of tight edges. A procedure for doing so is described in Alg. 2. First, the layering is initialized. This can be done by selecting the source nodes and assigning them to the first layer, then selecting nodes whose all predecessors are already assigned to a layer and assigning them to the second layer and so on … This layering is clearly feasible. Then we initialize the tree by adding all nodes which are reachable

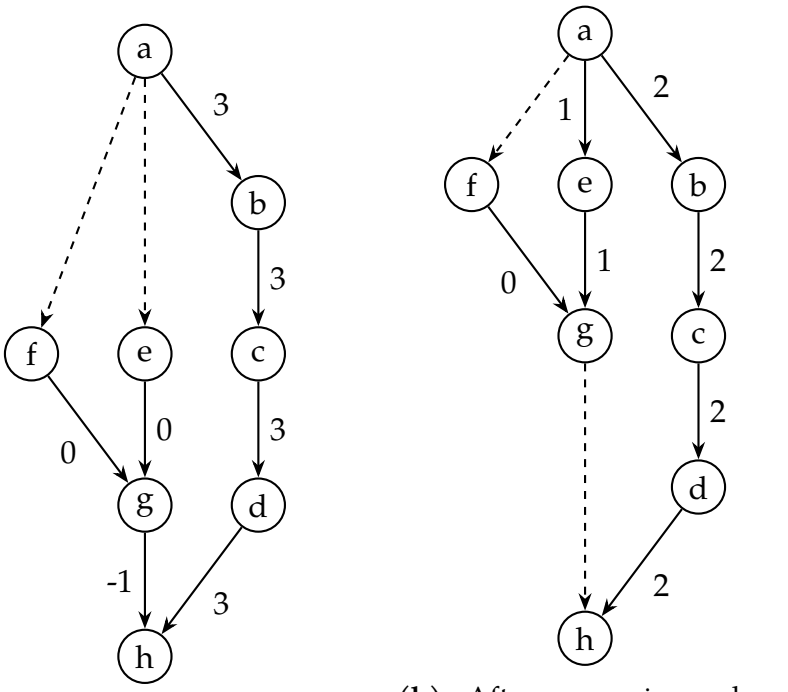
from the root (which can be arbitrarily chosen) through only tight edges. This tree is feasible since it was constructed from a feasible layering, but it is not necessarily a full spanning tree.

To construct the rest of the tree, we repeatedly find an edge which connects a tree node with a non-tree node with the minimum amount of slack. Then we can move the nodes in the tree either up or down (depending on whether the start or the end node are part of the tree) to make the edge tight. This operation only changes the slack of edges going between a tree node and a non-tree node. Since out of these we chose an edge with the minimal amount of slack, the tree still remains feasible and we can add the node to the tree. We repeat this process until the spanning tree contains all nodes.

2.3.2 Cut values

Given a feasible tree we can associate a *cut value* with each edge in the tree as follows. Removing a tree edge (u, v) splits the tree into two parts, the source component containing the source vertex u and the end component containing the end vertex v. The cut value of this edge is the number of edges that go from the source component to the end component (including (u, v)) minus the number of edges that go from the end component to the source component. In other words, it is the difference between the number of edges connecting the two components with the same direction as (u, v) and the number of edges with the opposite direction.

A negative cut value means more edges are going in the opposite direction, which indicates that there is a possibility to improve the layering by moving vertices in one of the components so that we make the opposite edges shorter and the edges with the same direction as the tree edge longer. The value by which we can move the component is the minimum amount of slack among the opposite edges. Otherwise, the layering would become infeasible. After moving the component, the tree edge with the negative cut value is no longer tight, but the opposite edge with the least amount of slack became tight and we can use it as a new tree edge. If the minimal slack is zero no improvement is made. However, if it is greater than zero we improve the total edge length, since more edges were shortened then lengthened. The tree



(**a**) Initial tree

(**b**) After swapping edges (g, h) and (a, e)

Figure 2.2: Cut values (from [3])

edge which was removed is called the *leaving edge* and the opposite edge which replaced it is the *entering edge*.

Fig. 2.2 gives an example of this operation. Fig. 2.2a shows the initial spanning tree (non-tree edges are dashed) and the cut values of its edges. Edge (g, h) has cut value -1, because it is the only edge going from its source component (consisting of vertices e, f, and g) to its end component, but there are two edges (a, f) and (a, e) going in the opposite direction. We can move the source component up by one layer, which makes the two edges tight and decreases the total edge length by 1. One of these edges can then be chosen as a new tree edge, for example (a, e). Note that this changes the cut values of the other edges in the tree. The result of this transformation is in 2.2b.

Algorithm 3: Network simplex algorithm (from [3])

```
Procedure network_simplex(G) begin
    tree ← feasible_tree(G)
    initialize cut values
    while there is a tree edge with negative cut value do
        e ← an edge with negative cut value
        f ← an edge with smallest slack among edges opposite to e
        move one component of e by slack(e)
        remove e from tree and add f
        update cut values
    normalize the layering
    introduce dummy vertices
```

2.3.3 Network simplex

The network simplex algorithm itself is described in Alg. 3. First, we obtain the initial feasible tree and calculate the initial cut values of all tree edges. Then we execute the main loop until there is no tree edge with a negative cut value, signifying no more improvements are possible. In each iteration, we first pick one of the edges with a negative cut value as the leaving edge. Then between the opposite edges we pick the one with the least amount of slack as the entering edge. We move one of the components to make the entering edge tight by adding or subtracting its slack (similarly as in Alg. 2) from the layer of all nodes in the component. Then we exchange the entering edge with the leaving edge updating the tree and the cut values. After the main loop terminates, we need to normalize the layering such that the least layer is zero and convert it to a proper layering by the procedure described in section 2.1.

2.4 Crossing reduction

For this phase, I use the standard layer-by-layer sweep and as the heuristic for one-sided crossing minimization I use a similar approach as Gansner et al. [3]. They combine a modified version of the median heuristic with a transposing heuristic to improve its results. However,

instead of the median heuristic I chose the barycenter heuristic because it is easier to implement and performs better [17].

The crossing reduction algorithm uses the fact that the layer assignment step produces a proper layering. That enables us to go layer by layer and each time reduce crossings only between two adjacent layers instead of minimizing crossings globally. Although this doesn't reduce the time complexity of the problem, it makes it easier to understand and implement.

In general, the layer-by-layer sweep algorithm first computes arbitrary initial ordering. Then it sweeps through the layers alternating between downward and upward sweeps. A downward sweep starts with the second layer and while holding the ordering in the first layer fixed, it reorders the second layer to reduce the number of crossings between them. Then the third layer is reordered while holding the second layer fixed. This continues until all layers except the first one are processed. The upward sweeping is symmetrical.

2.4.1 Counting crossings

The algorithm should terminate when no further improvements are possible or when the improvements get sufficiently small. To determine the improvement made by one iteration we need to be able to count the total number of crossings in a hierarchy.

The number of crossings between two layers is dependent only on the relative ordering of the vertices in those layers. Two edges (u_1, v_1) and (u_2, v_2) between the same layers cross if either $pos(u_1) < pos(u_2)$ and $pos(v_1) > pos(v_2)$ or if $pos(u_1) > pos(u_2)$ and $pos(v_1) < pos(v_2)$. Using this fact, we can count the number of crossings simply by iterating over all pairs of edges between the two layers and checking if they cross.

Since the input to the crossing reduction step is a proper hierarchy, where each edge connects only two adjacent layers, we can calculate the total number of crossings in the hierarchy by summing up the number of crossings between all adjacent layers.

Algorithm 4: crossing reduction (based on [3])

```
fails ← max_fails
best ← initial ordering of the layers
iter ← 0
while fails > 0 do
    barycenter(hierarchy, iter)
    transpose(hierarchy)
    if count_crossings(ordering) < count_crossings(best) then
        best ← ordering
        fails ← max_fails
    else
        fails ← fails − 1
    iter ← iter + 1
```

2.4.2 The algorithm

Alg. 4 describes the main part of the algorithm. We iteratively apply the barycenter and the transpose heuristic to the current ordering and if the number of crossings decreases, we save the new best ordering. The algorithm terminates when the number of consecutive iterations without an improvement exceeds `max_fails`.

The procedure *barycenter* is described in Alg. 5. Depending on the parity of the iteration we either perform a downward or an upward sweep. During the downward sweep we iterate from the second layer to the last and each time apply the barycenter heuristic (the upward sweep is analogous).

The heuristic is based on the idea that if we place each vertex as close to its neighbors as possible the edges incident on the vertex will be shorter and thus it will be less likely that they will cross other edges. To place vertices close to their neighbors, we calculate a weight for each vertex equal to the average position of its neighbors on the preceding layer (for the upward sweep on the next layer). Formally the weight is:

$$weight(u) = \frac{1}{d^-(u)} \sum_{v \in N^-(u)} pos(v)$$

Algorithm 5: barycenter heuristic (based on [3])

```
procedure barycenter(hierachy, iteration) begin
    if iteration is even then // downward sweep
        for i ← 1 to h − 1 do
            foreach u ∈ L_i do
                u.weight ← average position of N⁻(u)
            sort L_i based on vertex weights
    else // upward sweep
        // This case is symmetrical to the downward sweep
```

If a vertex has no neighbors the weight is set to its current position. After the weights are assigned we use them to sort the vertices on the current layer, thus obtaining the new ordering.

After the barycenter heuristic, we apply the transpose heuristic also known as the greedy switch heuristic. It repeatedly goes through all layers and greedily swaps pairs of adjacent vertices until no more improvements are possible.

To determine if swapping two vertices is beneficial the heuristic uses crossings numbers. The *crossing number* c_{uv} is the number of

Algorithm 6: transpose heuristic (from [3])

```
procedure transpose(hierachy) begin
    improved ← true
    while improved do
        improved ← false
        for k ← 0 to h − 1 do
            for i ← 1 to |L_k| − 1 do
                s ← L_k[i − 1]
                t ← L_k[i]
                if c_ts < c_st then
                    swap s and t
                    improved ← true
```

crossings between edges incident on u and edges incident on v if u is to the left of v.

The heuristic also uses the fact than swapping two consecutive vertices u and v only affects the number of crossings between edges incident on those two vertices and has no effect on crossings with any other edges. Consequently, if u is immediately before v and $c_{vu} < c_{uv}$ it is beneficial to swap the vertices.

The *transpose* procedure is described in Alg. 6. The main loop iterates until no improvement is made by the last iteration. In each iteration we go through the layers from top to bottom and through each pair of adjacent vertices from left to right. We calculate the two crossing numbers and if an improvement can be made we swap the vertices.

2.4.3 Crossings between inner segments

For the next the next step of the Sugiyama framework it is important that no two inner segments cross (for the definition of an inner segment see section 2.1). This would cause undesirable bends in long edges. Thus, it is important that the crossing reduction phase eliminates all inner segment crossings.

The barycenter heuristic guarantees this property [18]. Assume that after the application of the *barycenter* procedure this property does not hold. That means there exist two inner segments (u_1, v_1) and (u_2, v_2) which cross. Without loss of generality we can assume that they cross because $pos(u_1) < pos(u_2)$ and $pos(v_1) > pos(v_2)$ and that they connect the same layers (otherwise they could not cross). Since they are inner segments, u_1 is the only predecessor of v_1 and u_2 is the only predecessor of v_2, so during the downward sweep the weight of v_1 is $pos(u_1)$ and the weight of v_2 is $pos(u_2)$. Which means that after the sorting step $pos(v_1) < pos(v_2)$ which is a contradiction. The case for the upward sweep is analogous.

If the *transpose* procedure is applied to an ordering which satisfies the property that no inner segments cross, it preserves this property. Assume that this statement does not hold. Again, after executing the procedure there must be two inner segments (u_1, v_1) and (u_2, v_2) which cross and without loss of generality $pos(u_1) < pos(u_2)$ and $pos(v_1) > pos(v_2)$ and the segments connect the same two layers.

However, before the execution of the procedure either u_1 and u_2 or v_1 and v_2 must have been in the reverse order and at some point they must have been swapped since originally there were no crossing inner segments.

Assume that u_1 and u_2 were the ones originally in the reverse order ($pos(u_2) < pos(u_1)$) and they got swapped. The only way the algorithm could swap them is if $c_{u_2u_1} > c_{u_1u_2}$. By definition of an inner segment both u_1 and u_2 are dummy vertices. Additionally, each dummy vertex has both in-degree and out-degree one and thus only two incident edges. Each of the vertices has one incident inner segment and they don't cross, so depending on whether the other two incident segments cross or not $c_{u_2u_1} = 1$ or $c_{u_2u_1} = 0$. If the two vertices were swapped, then at least the two incident inner segments would cross, so $c_{u_1u_2} \geq 1$. That means $c_{u_2u_1} \leq c_{u_1u_2}$ which is a contradiction. The case for v_1 and v_2 getting swapped is analogous.

Since the crossing reduction algorithm always first applies the barycenter heuristic and then the transpose heuristic, it follows that if the algorithm finds at least one better ordering, the final ordering will have no crossing inner segments. However, if no improvement is made then the final ordering is the same as the initial one. To ensure that even in this case the property holds the initial ordering is computed using one sweep of the barycenter heuristic.

2.5 Coordinate assignment

The main focus of this section is the assignment of horizontal coordinates. Assigning the vertical coordinates is trivial since all vertices on the same layer have the same vertical coordinate. This coordinate can be determined by simply choosing a coordinate for the topmost layer and then assigning coordinates to the remaining layers based on the given minimum separation between layers.

For the assignment of x-coordinates I implemented the algorithm by Brandes and Köpf [16]. The main advantages of this algorithm are that it runs in linear time and guarantees that each edge has at most two bends.

2.5.1 General overview

To obtain readable drawings, the horizontal coordinates should be assigned according to several criteria [16].

(1) Positions of vertices should be balanced with respect to their neighbours.

(2) Edges should be short.

(3) Edges should be as straight as possible.

Since we are only interested in x-coordinates, the length of an edge segment (u, v) is defined as $|x(u) - x(v)|$. The length of edges leaving from vertex u can then be calculated as $\sum_{(u,v)\in E} |x(u) - x(v)|$. This term is minimized when $x(u)$ is the median of all $x(v)$. So to achieve item (2) we can align each vertex with its median neighbor, which achieves item (1) as well. The item (3) is accomplished by ensuring all dummy vertices are vertically aligned. Thus, all inner segments are vertical and edges have at most two bend points: one at the first dummy vertex and one at the last dummy vertex.

Each vertex can be aligned either with its upper or lower neighbors. Furthermore, we can consider the vertices for alignment on each layer either from left to right, which gives precedence to the vertices on the left, or right to left, which gives precedence to the vertices on the right. Thus, we can create four different layouts: one for each combination of upward or downward alignment and left-to-right (alignment to the left) or right-to-left (alignment to the right) order of considering vertices on their layers.

The algorithm consists of three steps. The first two steps are done four times and they produce the four different layouts. The first step, called *vertical alignment*, aligns vertices with their neighbors (based on the direction of the alignment). In the second step, called *horizontal compaction*, aligned vertices obtain the same x-coordinate and blocks of aligned vertices are shifted together to make the layout more compact. In the final step, the four candidate layouts are combined into the final layout.

In the following sections the algorithm is described in greater detail. However, only for the case of upward alignment to the left. The

other three cases are symmetric and the differences are pointed out at relevant parts of the algorithm.

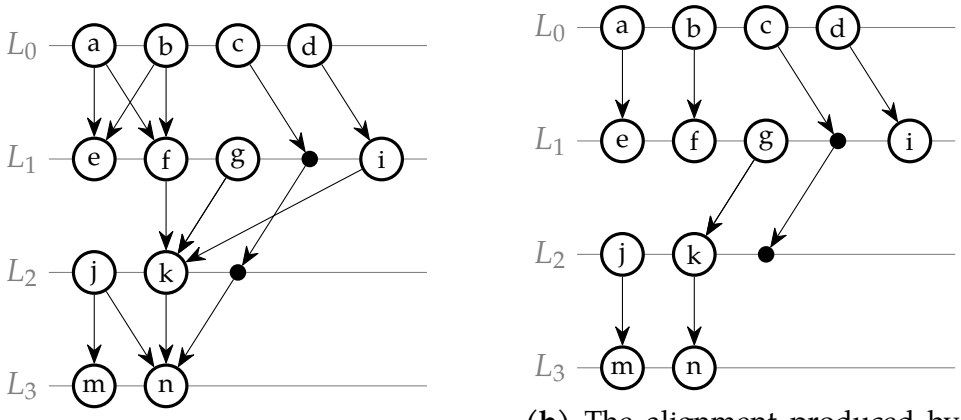
2.5.2 Alignment conflicts

Since we align each vertex with one of its neighbors, each alignment corresponds to an edge between two vertices. Two alignments conflict if their corresponding edge segments cross or if they share a vertex. In this situation it is impossible to perform both of the alignments. There are three types of conflicts which can arise.

Type 0 conflicts arise when two non-inner segments cross or share a vertex. These conflicts are resolved in favor of the alignment which is attempted first when aligning vertices on a particular layer.

Type 1 conflict occurs when a non-inner segment crosses an inner segment. To guarantee vertical inner segments, we always resolve this conflict in favor of the inner segment. To make checking for these conflicts more efficient, we find all non-inner segments participating in a type 1 conflict, and mark them during a preprocessing step.

Type 2 conflicts correspond to a pair of crossing inner segments. This conflict prevents one of them from being vertical which increases the number of bends. However, as discussed in section 2.4.3, the algorithm for crossing reduction guarantees that no inner segments cross, so we can safely assume no type 2 conflicts can occur.



(**a**) Original graph

(**b**) The alignment produced by Alg. 7

Figure 2.3: Upper alignment to the left (from [16])

Algorithm 7: Vertical alignment (from [16])

```
Procedure vertical_align() begin
    foreach u ∈ E do root[u] ← u
    foreach u ∈ E do align[u] ← u
    for L ← L_0 to L_{h−1} do
        last ← −1
        for v ← L[0] to L[|L| − 1] do
            if v has any predecessors then
                l ← left median predecessor of v
                r ← right median predecessor of v
                for u ← l, r do
                    if (u, v) not marked and pos(u) > last then
                        align[u] ← v
                        root[v] ← root[u]
                        align[v] ← root[u]
                        last ← pos(u)
                        break
```

2.5.3 Vertical alignment

To represent the alignment of vertices we use blocks. A block is a maximal set of vertically aligned vertices and its root is the topmost vertex. For each vertex we store a reference to its lower aligned neighbor (the lowest vertex in a block references the root) and a reference to the root of its block. This is the case for the upward alignment. For the downward alignment the root of a block is the lowest vertex and each vertex stores a reference to its upper aligned neighbor. The algorithm in Alg. 7 determines the alignment and populates this data structure.

The layers are traversed from top to bottom (reversed for the downward alignment) and on each layer vertices are traversed from left to right (reversed for the right alignment). For each vertex we consider the upper neighbors $v_1, \ldots, v_k$ sorted by their position on the preceding layer. We take $v_{\lfloor (k+1)/2 \rfloor}$ as the left median neighbor and $v_{\lceil (k+1)/2 \rceil}$ as the right median neighbor. We first try to align with the left median

neighbor and then if it fails with the right median neighbor (the order is switched for the right alignment). The alignment can fail either because the corresponding edge segment is marked as participating in a type 1 conflict (by a preprocessing step) or it conflicts with the previous alignment on the current layer.

Fig. 2.3 illustrates the upper left alignment. Fig. 2.3a shows the original graph (the dummy vertices are filled black) and Fig. 2.3b shows for each vertex the predecessor it was aligned to.

2.5.4 Horizontal compaction

During horizontal compaction, vertices in the same block are assigned the same x-coordinate and blocks are grouped into classes, which are shifted so that the drawing is as compact as possible. In Fig. 2.4 the blocks are highlighted in blue and classes in green. The root of each block is red.

Classes are defined as follows. Each block which contains only vertices which are the leftmost on their layers establishes a new class and its root becomes the root of the whole class. In Fig. 2.4 these would be the blocks $\{a, e\}$ and $\{j, m\}$. The class of a block B is then recursively determined as the class of the predecessor of B with the uppermost root. The predecessors of the block B are all the blocks which contain a vertex immediately before some vertex in B, this is illustrated using the arrows between blocks.

For the right alignment, we would similarly define and use the successors of a block. And for the downward alignment, we would use the lowest predecessor/successor to define a class of a block.

Classes defined in this way are free to move with respect to each other. In Fig. 2.4 we can see that the class containing the block $\{j, m\}$ can be shifted closer to the other class to make the drawing more compact.

The procedure itself is described in Alg. 8. First, we go through all blocks and calculate the position of their root relative to the root of their class. The position is determined using the procedure *place_block*, which places the root at the maximum coordinate of the preceding blocks, plus the minimum separation of two vertices denoted δ (for the right alignment it is placed at the minimum coordinate minus δ). It goes through all vertices w in the block with root v. If there is a

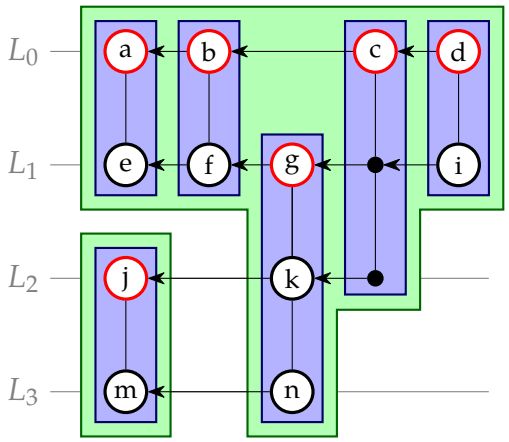


Figure 2.4: Blocks and classes (from [16])

block to the left of w, it is recursively placed. If it is a block from the same class, the position of the current block is updated. If the adjacent block is in a different class, we calculate the value by which the class needs to be shifted, so that it is as close to the current class as possible. After the relative coordinates are calculated, we can go through each vertex and assign it the coordinate of the root of its block shifted by the appropriate value.

2.5.5 Balancing

As the last step, we combine the four layouts produced by the previous two steps of the algorithm into the final layout. Before doing this, the layouts have to be aligned with the layout with the smallest width. The layouts aligned to the left are shifted so that their minimum coordinate agrees with the minimum coordinate of the smallest width layout and the layouts aligned to the right are shifted to make the maximum coordinates agree.

Then, using the four candidate coordinates $x_1 \leq x_2 \leq x_3 \leq x_4$, the final coordinate of each vertex is calculated as the average median: $\frac{1}{2}(x_2 + x_3)$. The average median guarantees that the order and the minimum separation of vertices is preserved [16]. The Alg. 9 gives the overview of the whole algorithm.

Algorithm 8: Horizontal compaction (from [16])

```
procedure horizontal_compaction() begin
    foreach v ∈ V do
        class[v] ← v
        shift[v] ← inf
        x(v) ← undefined
    foreach v ∈ V do
        if root[v] = v then place_block(v)
    foreach v ∈ V do
        x(v) ← x(root[v])
        if shift[class[root[v]]] < inf then
            x(v) ← x(v) + shift[class[root[v]]]

procedure place_block(v) begin
    if x(v) is already defined then
        return
    x(v) ← 0
    w ← v
    repeat
        if pos(w) > 0 then
            u ← root[prev(w)]
            horizontal_compaction(v)
            if class[v] = v then class[v] ← class[u]
            if class[v] ≠ class[u] then
                shift[class[u]] ← min(shift[class[u]], x(v) − x(u) − δ)
            else
                x(v) ← max(x(v), x(v) + δ)
        w ← align[w]
    until w = v
```

2.6 Edge routing

This is the last step of the drawing algorithm, which is executed after the Sugiyama framework. During this step, the final control points

Algorithm 9: x-coordinate assignment (from [16])

```
mark type 1 conflicts
for vertical direction in up, down do
    for horizontal direction in left, right do
        vertical_alignment()
        horizontal_compaction()
align to layout with smallest width
calculate the coordinates as the average median
```

of edges are calculated such that no edge intersects a node, and the reversed and removed edges are restored.

The edges (meaning edges in the original graph before they were subdivided by dummy vertices) are drawn as poly-lines which are represented as a list of points. The first and the last point are called ports and they are the points where the edge connects to the border of its adjacent nodes. The "middle" points are the bend points of the edge and they are simply the coordinates of the dummy nodes into which the original edge was decomposed.

The reversed edges are restored by reversing the list of points representing the edge. The edges which formed two-cycles and were removed are reinserted by marking the other edge forming the two-cycle which was left in the graph as "bidirectional".

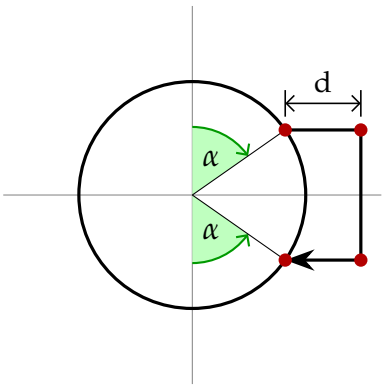


Figure 2.5: Loop control points

For the loops we need to calculate entirely new control points. The way these points are calculated is illustrated in Fig. 2.5. The shape of the loop is governed by two parameters: an angle α with the vertical axis going through the middle of the node and a horizontal distance d. The angle is used to obtain the ports and the loop is completed by inserting a point d units to the right of both of the ports.

2.6.1 Ports

A simple and naive way to compute the ports for an edge is to find the point where the first/last segment of the edge would intersect the starting/ending node if the ports were in the center of the nodes. This method gives good looking results. However, it can cause the edge to intersect a neighboring node if the next control point (either a dummy node or a regular node) is too far or the sizes of the nodes are large as can be seen in Fig. 2.6.

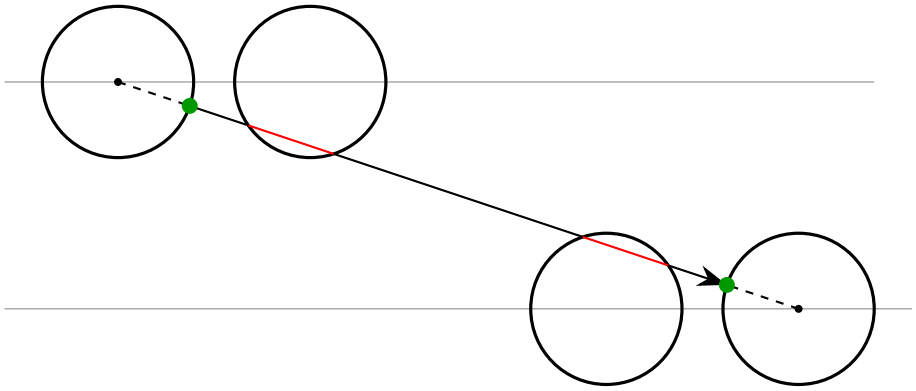


Figure 2.6: Intersections caused by bad port positioning

There are several solutions to this problem. One would be to introduce new bend points in the edge to go around the node. However, this might cause it to cross other edges and we would in turn need to introduce new bend points in them as well. As bends reduce readability, this is undesirable.

Another solution is to increase the distance between the layers connected by the edge until the edge no longer intersects the node. But if the distance between two adjacent layers in the drawing is significantly higher then between the other layers, the reader might

see the graph as comprised of two independent clusters, which might misdirect his attention and lead to wrong conclusions.

The way I compute the ports is similar to the naive method. I also calculate the port as the intersection of the node border and a line going from a point inside the node to the next control point. However, the starting point inside the node is not necessarily the center (although it is preferable if possible) but is shifted either up or down, based on the direction of the edge, as illustrated in Fig. 2.7.

Since all nodes have the same size, it is guaranteed that if the center point were shifted all the way to the border of the node there would be no intersections. However, to achieve better-looking drawings, it is desirable to find for each node the least shift possible. Each node has four distinct shift values, one for each of the possible edge directions.

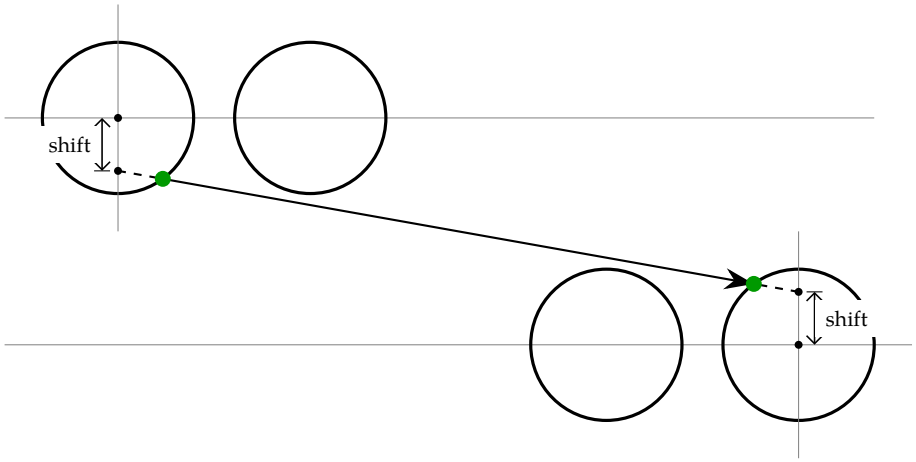


Figure 2.7: Shifting center points to eliminate intersections

To find the least shifts I go through each non-inner segment (all inner segments are vertical and thus cannot intersect any node). If the segment intersects any nodes I take the current shifts of its endpoints and iteratively increase them until it doesn't cross any nodes.

The shifts are calculated for dummy nodes as well as regular nodes. Since dummy nodes have no border, the shift is not used to compute a port but instead the shifted point is taken as an additional bend point. This can cause unwanted crossings with segments incident on adjacent dummy nodes. To prevent this, the shifts in each continuous sequence of adjacent dummy nodes are taken as the maximum shift among

them and the regular nodes enclosing this sequence are updated with this value as well.

3 The library

In this chapter, I describe the library I implemented for this thesis. I go over its interface and show how it can be used to implement a simple application for producing SVG images of graphs.

3.1 Interface

It is a C++ header-only library. It can be used by including the header file `interface.hpp` which includes all the other necessary headers. The advantage of this approach is that it is extremely simple to use and distribute and requires no linking against external targets. The only dependency is the C++ standard library and it requires the C++17 standard.

It provides two basic functionalities. The first one is incrementally building a graph. This can be done by incrementally adding vertices and edges connecting those vertices. The second basic functionality is creating a representation of this graph in 2D space.

3.1.1 Describing a graph

The graph is represented using the `graph` class. It provides two methods for adding elements to the graph:

```
vertex_t add_node();
void add_edge(vertex_t u, vertex_t v);
```

The `add_node` method adds a new vertex to the graph and returns its identifier of type `vertex_t`, which is an implementation-defined integral type. An edge between two vertices with identifiers u and v can be added using the `add_edge` method. The behavior of this method is undefined if the same edge is added twice or if an identifier other than one returned by `add_node` is used.

3.1.2 Creating a layout

The `sugiyama_layout` class represents the layout of a graph in space. The layout itself is computed during a construction of an object of

this class which then provides the calculated positions of the vertices and the edges through its interface. The class provides two different constructors:

```
sugiyama_layout(graph g);
sugiyama_layout(graph g, attributes attr);
```

Since the input graph is modified by the algorithm, it is always taken by value to enable the user to construct two different layouts for the same graph or construct a second layout with only minor modification of the input graph without rebuilding it from scratch. If the user only wants to make one layout for a given graph and doesn't want to incur the extra overhead of a copy, the graph can be moved into the constructor.

The `attributes` object contains the parameters describing the positioning of elements in the drawing. If it is not provided, one with the default values for these parameters is used. It contains the following members:

```
struct attributes {
    float node_size;
    float node_dist;
    float layer_dist;
    float loop_size;
    float loop_angle;
};
```

The `node_size` member is the radius of the vertices and applies to all vertices in the drawing, `node_dist` is the minimum distance between the borders of two nodes (not their centers) and similarly `layer_dist` is the minimum distance between two adjacent layers. The distance is measured from the lower border of nodes on the upper layer to the upper border of nodes on the lower layer. Lastly, `loop_size` and `loop_angle` describe the size of loops (see section 2.6).

3.1.3 Using the layout

After the `sugiyama_layout` object is constructed, it contains the description of the layout of the graph. A point (or a vector) in the 2D plane is represented with the `vec2` struct.

```
struct vec2 {
    float x, y;
};
```

The bounding box which encloses the whole graph can be obtained with:

```
float width() const;
float height() const;
vec2 dimensions() const;
```

To obtain the positions of nodes one can use the following method:

```
const std::vector<node>& vertices() const;

struct node {
    vertex_t u;
    vec2 pos;
    float size;
};
```

The `vertices` method returns a collection of positions of all the vertices described by the `node` structure where `u` is the identifier of the corresponding vertex, `pos` the position of its center and `size` its radius. The returned collection can be indexed using node identifiers, so the expression `layout.vertices()[v]` can be used to obtain the `node` corresponding to the vertex with identifier `v`.

```
const std::vector<path>& edges() const;

struct path {
    vertex_t from, to;
    std::vector<vec2> points;
    bool bidirectional;
};
```

The control points of edges can be obtained in a similar way as vertices using the `edges` method. The `path` structure contains the identifiers of the endpoints of its associated edge, the points defining the poly-line corresponding to the edge and a flag indicating if the edge is bidirectional.

3.2 Sample application

To show how the library can be used I implemented a simple command line application for creating SVG images of graphs. It can be used to draw a graph described in a limited subset of the DOT format. It can take either a file and produce an SVG image of the graph it represents or it can take a path to a directory and draw all the files with the `.gv` extension (the extension of files in DOT format) it contains. The command line usage is: `./draw [-d] source destination`. If the flag `-d` is given, the source and destination are interpreted as directories, otherwise as files.

3.2.1 Input format

As mentioned earlier, the graph is described in a subset of DOT language. The input file has to start with the `digraph` keyword and the whole contents be enclosed in curly braces. The only purpose of this is to be able to draw the same file with *dot*.

There are several graph attributes in the form of key-value pairs which can be used to control the appearance of the elements in the drawing. These include `nodesize` for setting the size (radius) of nodes, `ranksep` and `nodesep` for setting the layer separation and node separation respectively, `loopsize` and `loopangle` for customizing loops and `fontsize` for changing the size of the font used for the node labels. The values for all of them are specified in inches except for the `fontsize` which is in points (pt) and `loopangle` which is in degrees[1].

The structure of the graph itself is described using edge declarations in the form `from -> to` where `from` and `to` are identifiers of nodes. Notably, the application doesn't support any node declarations. Thus the identifiers are used as labels and the nodes are implicitly created by the edge declarations. The following is an example of a valid graph description:

```
digraph {
    ranksep=0.3;
    nodesep=0.3;
    nodesize=0.2;
```

1. The units were chosen to preserve compatibility with *dot*

```
    fontsize=10;
    loopangle=60;
    loopsize=0.2;
    a -> b;
    b -> c;
    b -> d;
}
```

3.2.2 Creating the graph

There are two parts of the implementation I want to highlight, since these are the parts most relevant to the usage of my library. These are creating the graph representation from the edge declarations, and using the interface of `sugiyama_layout` to produce an image.

When parsing the edge declarations, we have two sets of node identifiers: the ones used in the input file, which I will refer to as labels, and the ones used by the `graph` class. To keep track of which label corresponds to which identifier, we need to keep a map between them. Preferably, we need one map for each direction, since to add an edge we need to convert from labels to identifiers and later when drawing the graph we need to know which identifier has which label. The next code snippet shows how a new edge can be added using the labels `from` and `to` and the two maps `to_id` and `to_label`.

```
if (!to_id.contains(from)) {
    vertex_t u = graph.add_node();
    to_id[from] = u;
    to_label[u] = from
}
if (!to_id.contains(to)) {
    vertex_t u = graph.add_node();
    to_id[to] = u;
    to_label[u] = to
}
graph.add_edge(to_id[from], to_id[to]);
```

We need to check if the labels have been encountered before and if not we need to add the new node to the graph and create the mappings. Then we can simply add the new edge.

3.2.3 Creating the image

To create the SVG file, I use my own SVG writer class. It provides only the necessary functionality to draw a graph. The edges are drawn using the `polyline` element, the nodes and their labels are drawn using the `circle` and `text` elements and the arrows using the `polygon` element. Using this functionality, it is easy to convert the representation of a graph drawing provided by the `sugiyama_layout` class into SVG. The process is illustrated in the following code snippet.

```
svg_img img(filename);
for (auto node : layout.vertices()) {
    img.draw_circle(node.pos, node.size);
    img.draw_text(node.pos, to_label[node.u]);
}
for (const auto& path : layout.edges()) {
    img.draw_polyline(path.points);
    img.draw_arrow(path.points.back());
    if (path.bidirectional)
        img.draw_arrow(path.points.front());
}
```

We go through the nodes and draw them as circles with their labels in the middle. Then we iterate through the paths and draw them as poly-lines with an arrow at the last point and if the edge is bidirectional at the first point as well.

4 Comparison with dot

In this chapter, I compare the images produced by the application described in the previous chapter with the images produced by *dot*. First, I go over the main differences on several handpicked graphs. Then I perform a more thorough comparison using data collected from four graph sets, each consisting of 100 randomly generated graphs.

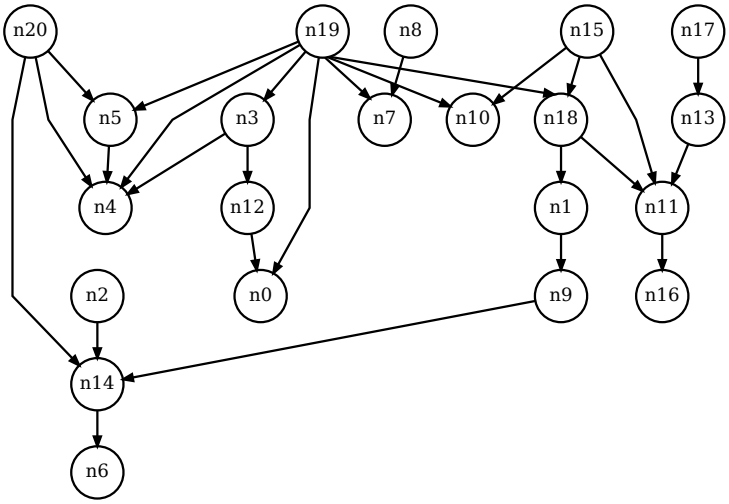
4.1 Example graphs

To make the comparison as objective as possible, I made the visual style of the drawings as close as possible. That includes setting the same values for parameters which control the size and positioning of elements in the drawing, such as node separation, layer separation, or the size of nodes. The shape of nodes needs to be matched as well since *dot* draws nodes as ellipses by default, as opposed to circles. Furthermore, I used the same font size and the same thickness of edges and node borders as *dot*.

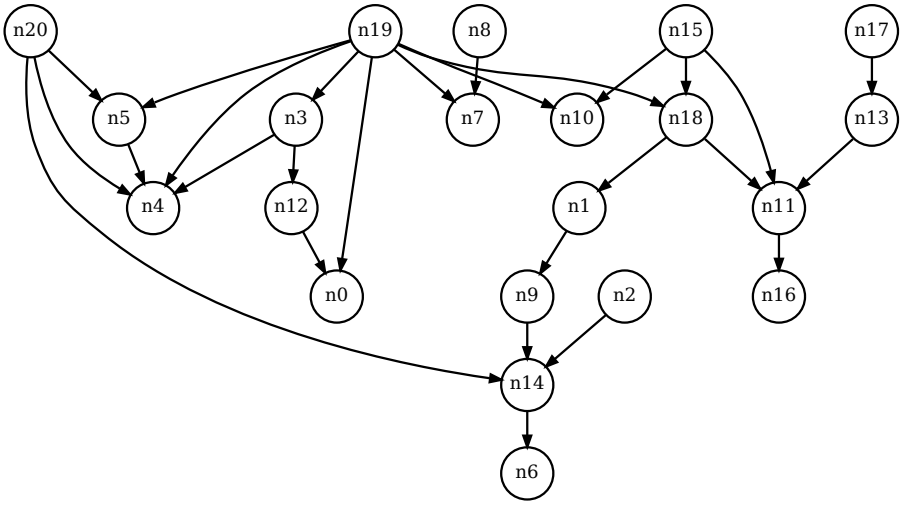
Fig. 4.1 shows the first example graph. The most obvious difference between the drawings is that *dot* draws edges as curves, which makes the drawing more aesthetically pleasing. However, if we look away from this fact, the drawings look fairly similar. The layering is precisely the same and they contain the same number of edge crossings. The positions of nodes are well balanced in both drawings, but my drawing is slightly narrower since the nodes are positioned more compactly.

In general, drawings where edges are drawn as curves are more aesthetically pleasing than drawings with poly-line edges. Additionally, to asses the quality of a poly-line edge routing algorithm, it is better to compare it to another such algorithm. Thus, in all following comparisons I will use *dot* drawings where the edges are drawn as poly-lines.

Fig. 4.2 shows the same graph as in Fig. 4.1 but with poly-line edges. We can see that the edges try to approximate the curves which requires more bends. An example of this is the edge $(n20, n14)$. My implementation places greater emphasis on vertical edges with fewer bends. Another difference is that *dot* sometimes merges the first part of several edges to avoid an intersection with a node as can be seen



(**a**) My library



(**b**) Dot

Figure 4.1: Example graph

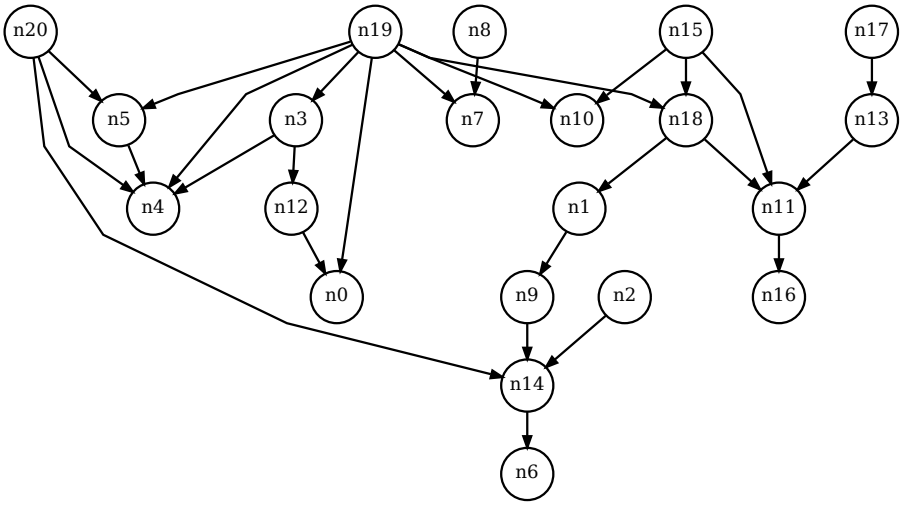


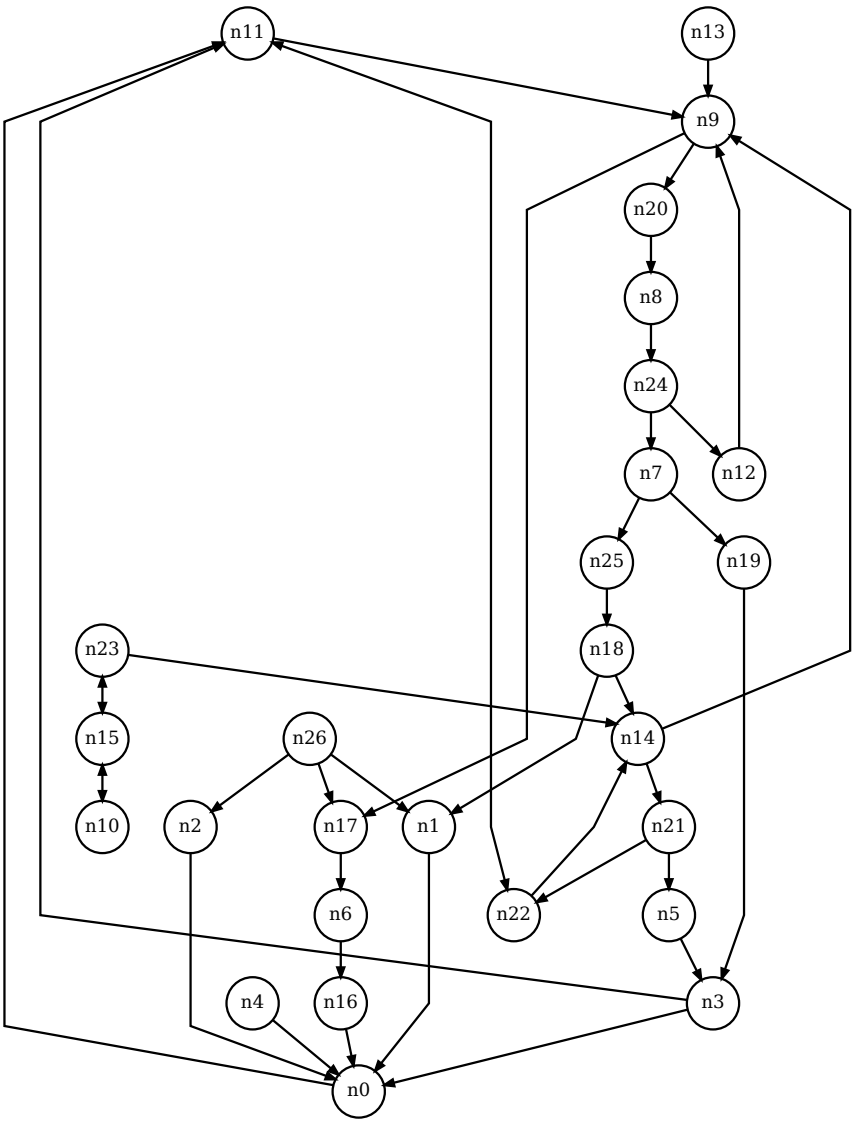
Figure 4.2: A drawing by dot with poly-line edges

with the edges $(n19, n10)$ and $(n19, n18)$. This cannot happen in my library. If it is possible, *dot* also routes edges through a more direct path and eliminates bends as is the case with the edge $(n19, n0)$.

The next example graph is in Fig. 4.3. It contains several cycles which means some edges need to point against the flow of the drawing. We can see that the drawings differ in their handling of bidirectional edges. In my drawing, the edge is drawn as a single poly-line with arrows on both ends. On the other hand, *dot* calculates control points for two different poly-lines, each for one of the directions. There does not seem to be a significant difference in the number of reversed edges.

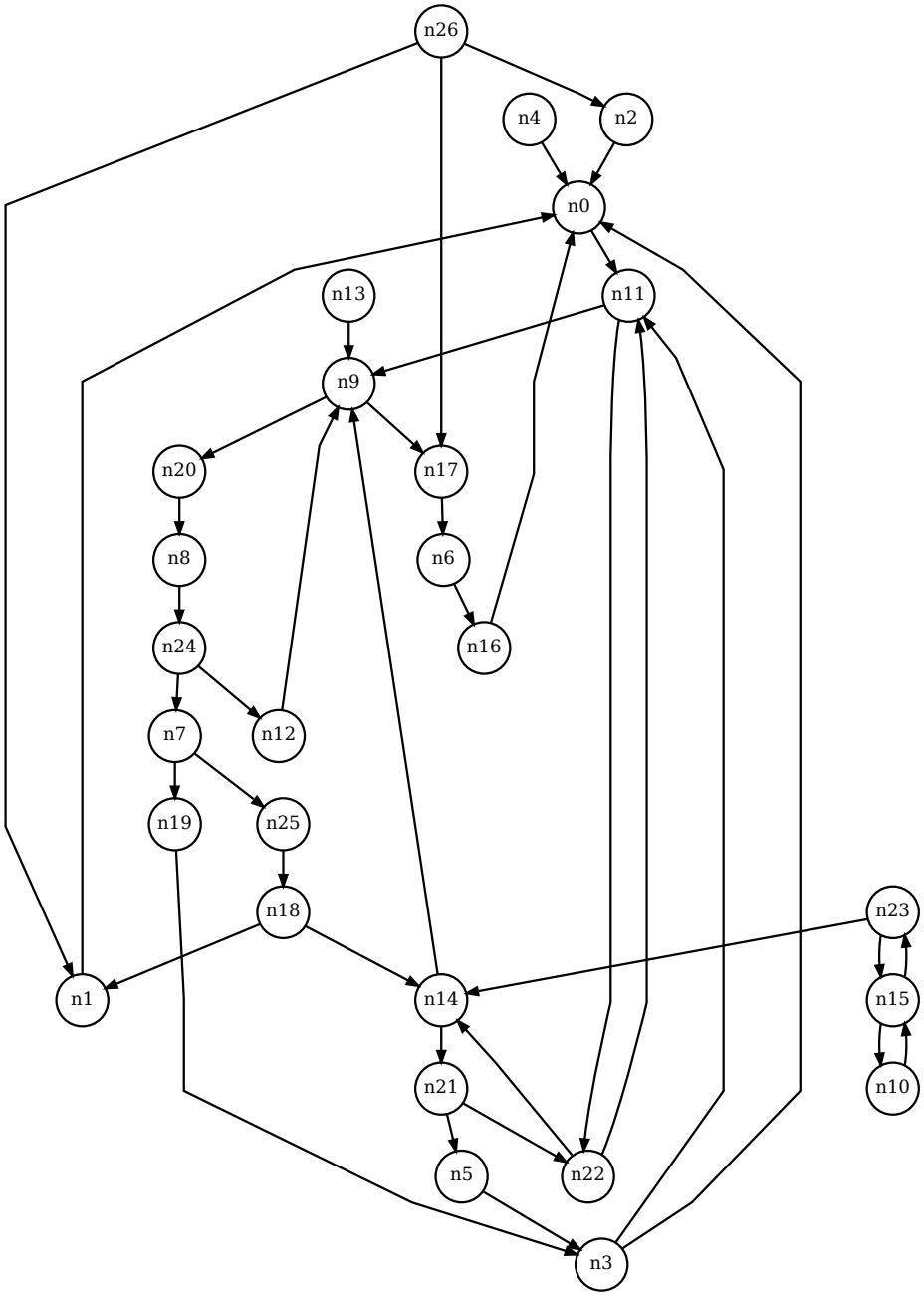
As in the previous example, the other aspects of the drawings are comparable as well. The number of crossings is almost the same, the positions of nodes are well balanced, and the drawings have similar dimensions.

Fig. 4.4 highlights some additional differences in edge routing. In this image, we can see the different methods for avoiding an intersection between a node and an edge. In my drawing, it is achieved by shifting the ports. However, in the *dot* drawing it is accomplished by adding more bends and even merging parts of edges. An advantage of the approach used by *dot* is that the last segment of the edge can



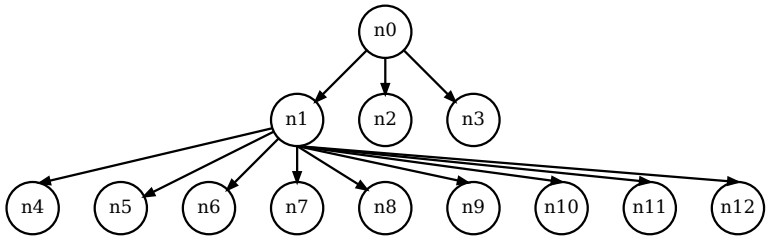
(**a**) My library

Figure 4.3: Graph with reverse edges

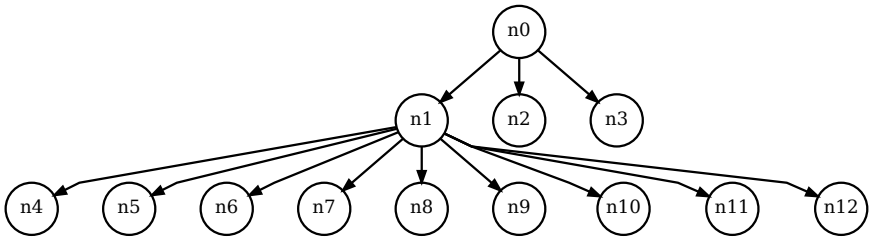


(**b**) Dot

Figure 4.3: Graph with reverse edges (cont.)



(**a**) My library



(**b**) Dot

Figure 4.4: Differences in avoiding edge-node intersections

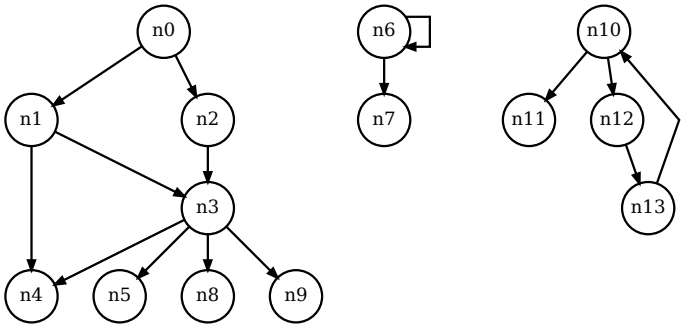
always aim straight to the center of the node, which makes the arrows look more natural.

Lastly, Fig. 4.5 shows how a disconnected graph is drawn. Both drawings use the same basic idea of putting the connected components in a row next to each other. However, *dot* puts them closer together, so the drawing is more compact. Since my library takes into account only the bounding boxes of the components, they are more spread out.

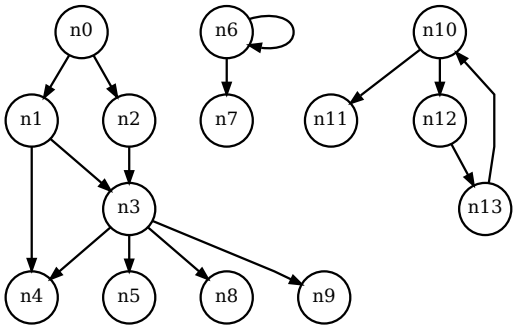
4.2 Statistics

In this section, I perform a more thorough comparison of my library and *dot*. For this purpose, I created a set of test graphs and measured several different parameters in the drawings of these graphs:

- Number of edge crossings
- Total edge length



(**a**) My library



(**b**) Dot

Figure 4.5: Handling of disconnected graphs

- Number of bends in edges
- Number of reversed edges

They are based on the aesthetic criteria described in section 1.3. Lower values of these measures generally mean the drawing is more aesthetically pleasing and readable.

4.2.1 The graph sets

I created four sets of graphs with 20, 50, 100 and 150 vertices. Each of the sets consists of 100 randomly generated graphs.

To generate them I used the *DAGmar* [19] graph generator[1]. It generates random directed acyclic graphs by picking them uniformly from the set of all possible graphs satisfying the input parameters, such as the number of nodes and edges. It produces graphs in the GraphML format. To be able to use them with *dot*, I converted them into the DOT format using the *graphml2gv* utility provided by *Graphviz*.

Each of the four sets of graphs were generated with edge density 1.4, which means the number of edges is $1.4 \cdot |V|$. The exact command I used to generate the graphs was:

```
java -jar DAGmar.jar multi -n "vertex_count"\
    -d "1.4" -i "1 to 100" -flat -c\
    -f "graph" target_dir
```

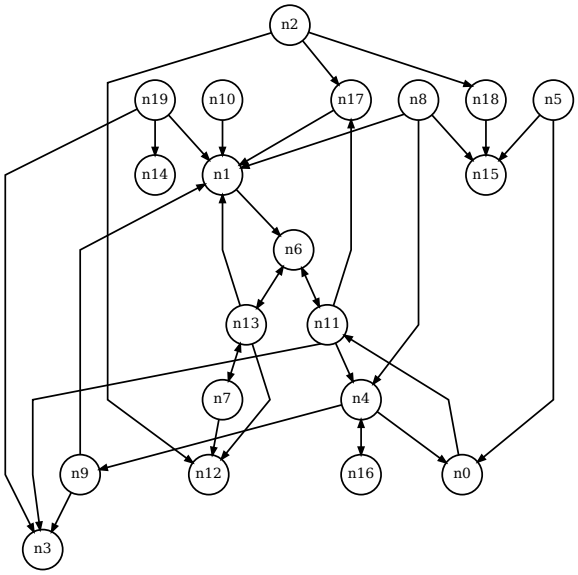
The resulting graphs are acyclic. Thus, to be able to compare the number of reversed edges, we need to introduce some cycles into the graphs. To do this, I added several edges to each graph by picking a random vertex and adding an edge to one of its predecessors. The number of edges added to each graph is $0.3 \cdot |V|$, so the final edge density of the graphs is 1.7. Fig. 4.6 shows an example of one of the final random graphs with 20 vertices.

4.2.2 Results

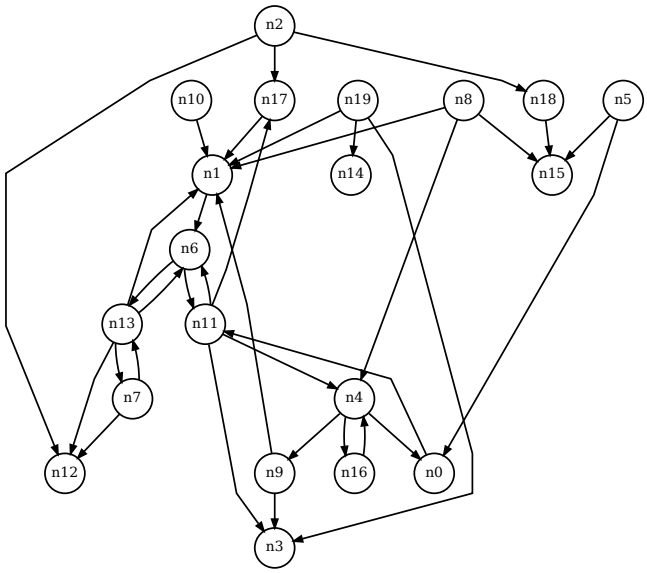
I created two images of each graph in the four data sets: one using the application described in section 3.2 and one using *dot*. From each of these images, I computed the four parameters: the number of edge crossings, the total edge length, the number of bends and the number of reversed edges. In this section, I present the collected data.

To visualize the distribution of values of a given parameter on a given data set I use box plots. The plots display five numbers describing a set of values: the minimum, the maximum, the first and the third quartile and the median. These values are depicted using a box with two whiskers extending from it. The whiskers show the minimum and the maximum. The bottom and the top of the box show the position of the first and the third quartile respectively and a line inside the box corresponds to the median.

1. The generator is available at `http://www.graphdrawing.org/data.html`



(**a**) My library



(**b**) Dot

Figure 4.6: Random graph with 20 vertices

The first set of plots in Fig. 4.7 compares the number of edge crossings. We can see that for all of the four graph sizes my library performs worse then *dot*. However, for smaller sizes the difference is not as large, so the readability of the drawings would be comparable. For larger graphs the difference is somewhat higher, but in large graphs the number of crossings is not as important [20].

Fig. 4.8 shows the comparison of the total edge length. We can see a similar trend as with the number of crossings. For small graphs, the edge length in my drawings is the same or slightly higher. However, for larger graphs, the difference increases. This is related to the data shown in Fig. 4.9 which displays the differences in number of bends in edges. We can see that my library produces fewer bends. The differences get progressively larger and for large graphs they are quite significant.

This probably happens because (as mentioned in section 4.1) *dot* routes edges through a shorter path and tries to approximate a curve

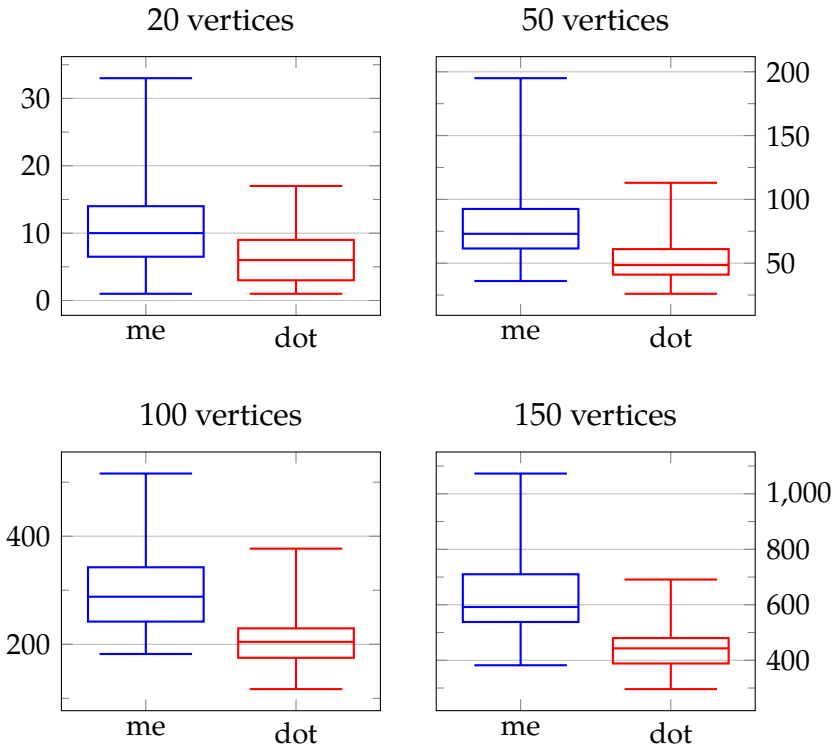


Figure 4.7: Comparison of number of edge crossings

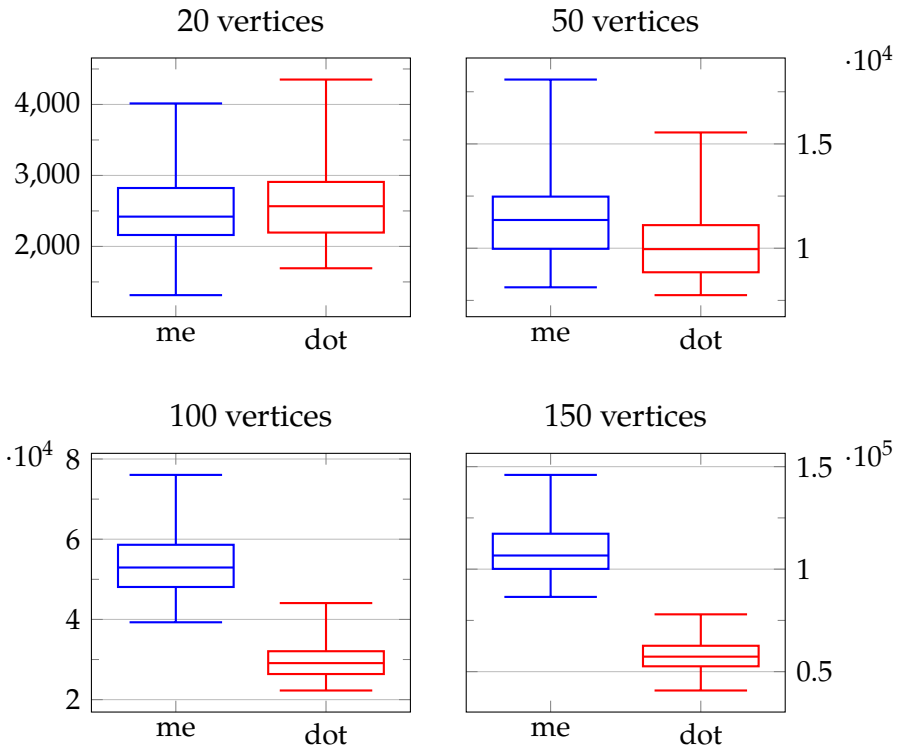


Figure 4.8: Comparison of total edge length

which requires more bends but decreases the edge length. It also introduces more bends when avoiding intersections between edges and nodes. On the other hand, in my library I use a node positioning algorithm that emphasizes vertical edges without bends which means they might take a longer path and make the total edge length longer. Another factor leading to fewer bends in my drawings is that the edge-node intersections are eliminated without introducing unnecessary bends.

Finally, the number of reversed edges is compared in Fig. 4.10. The plots show that in terms of reversed edges the drawings are nearly identical. This is not a surprise, since both *dot* and my library use a DFS based approach for cycle removal.

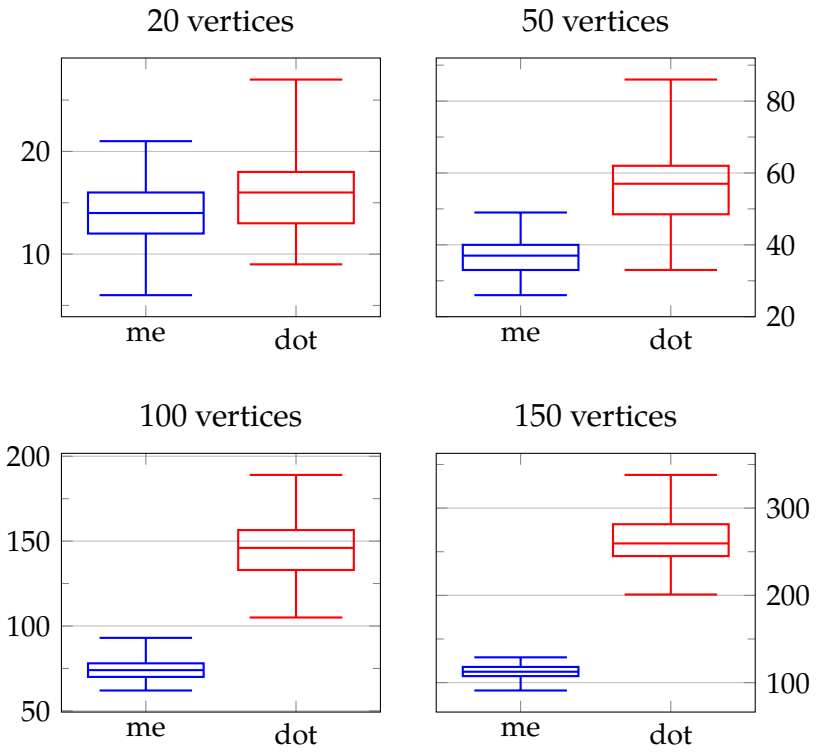


Figure 4.9: Comparison of number of bends in edges

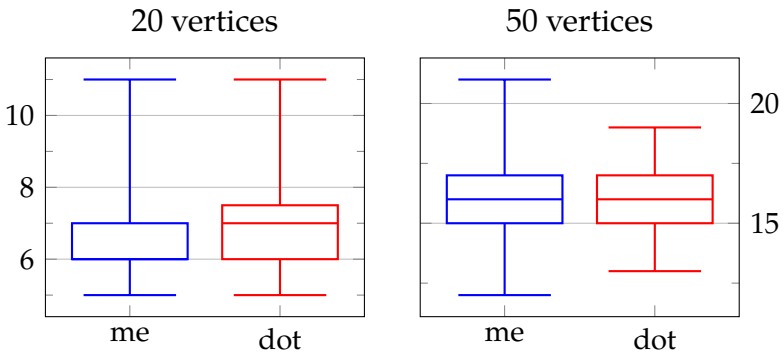


Figure 4.10: Comparison of number of reversed edges

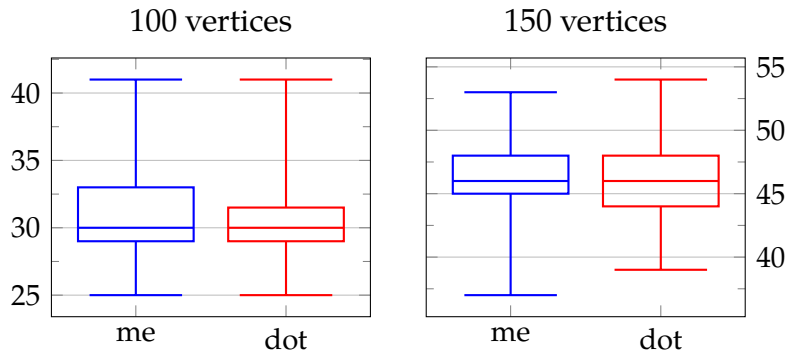


Figure 4.10: Comparison of number of reversed edges (cont.)

4.3 Summary

As shown in the previous sections, for small graphs the drawings produced by my library are of similar quality as poly-line drawings produced by *dot*. As the graphs get larger, the differences between the drawings increase. For large graphs, my drawings fall behind in terms of the number of edge crossings; however, crossing reduction is not as important for large graphs. My library also focuses on removing bends from edges which leads to higher total edge length and lower number of bends as opposed to *dot*.

5 Conclusion

The outcome of this thesis is a C++ header-only library for creating hierarchical drawings of directed graphs. It provides an interface for building a graph by incrementally adding vertices and edges and creates a layout of this graph in space. The layout is described by the positions of vertices, which are depicted as circles, and the positions of control points of edges, which are drawn as poly-lines. The layout can be controlled by setting various parameters such as the node radius or the minimum distance between two nodes. The library can be used to power graph-visualization applications.

To implement the library, I used a popular method for drawing directed graphs called the Sugiyama framework, which consists of four basic steps. For the cycle removal step, I used an algorithm based on DFS. For the layer assignment step, I implemented the network simplex algorithm. To reduce the number of edge crossings, I used the layer-by-layer sweep with a combination of the barycenter and transpose heuristics. Finally, for computing the final coordinates of vertices, I used an algorithm by Brandes and Köpf. I also implemented an algorithm for avoiding edge-node intersections based on shifting the edge ports.

To show how the library can be used, I implemented a command line application for creating SVG images of directed graphs which utilizes the library. It takes a file with a description of the input graph in a subset of the DOT language and produces a drawing as an SVG image.

I used this application to compare the images produced by my library with images produced by *dot*, which is a popular tool for drawing directed graphs. First, I highlighted the key differences on a few selected graphs. Then I presented the results of a more in-depth comparison done on four data sets, each consisting of 100 graphs. I concluded that for smaller graphs, the drawings are comparable, and for larger graphs, my images fall behind in certain areas (such as number of edge crossings and total edge length) but are superior in others (number of bends in edges).

5.1 Future work

There are several extensions which could be added to the library. In its current state all nodes have the same size so one possible extension is to add the ability to set the sizes of individual nodes. However, this would require changing the algorithm which eliminates edge-node intersections to accommodate for different sizes of nodes. Additionally, different shapes of nodes could be added as well.

Next, the crossing reduction algorithm could be improved to perform the same as *dot* even for large graphs. This would probably require a faster algorithm for counting edge crossings which would make it possible to perform more iterations of the employed heuristics.

Also, a new algorithm for edge routing could be used, which would route edges as curves. Finally, a feature for adding labels to edges would be useful.

Bibliography

[1] M. Kaufmann and D. Wagner, Eds., *Drawing Graphs: Methods and Models*, ser. Lecture Notes in Computer Science. Springer, 2001, vol. 2025, ISBN: 3-540-42062-2.

[2] H. C. Purchase, "Which aesthetic has the greatest effect on human understanding?", in *Graph Drawing, GD 1997*, G. DiBattista, Ed., ser. Lecture Notes in Computer Science, vol. 1353, Springer, 1997, pp. 248–261. DOI: 10.1007/3-540-63938-1_67.

[3] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo, "A technique for drawing directed graphs", *IEEE Trans. Softw. Eng.*, vol. 19, no. 3, pp. 214–230, 1993. DOI: 10.1109/32.221135.

[4] K. Sugiyama and K. Misue, "A simple and unified method for drawing graphs: Magnetic-spring algorithm", in *Graph Drawing, GD 1994*, R. Tamassia and I. G. Tollis, Eds., ser. Lecture Notes in Computer Science, vol. 894, Springer, 1995, pp. 364–375. DOI: 10.1007/3-540-58950-3_391.

[5] P. Eades, "A heuristic for graph drawing", *Congressus Numerantium*, vol. 42, pp. 149–160, 1984.

[6] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement", *Softw. Pract. Exp.*, vol. 21, no. 11, pp. 1129–1164, 1991. DOI: 10.1002/spe.4380211102.

[7] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs", *Inf. Process. Lett.*, vol. 31, no. 1, pp. 7–15, 1989. DOI: 10.1016/0020-0190(89)90102-6.

[8] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures", *IEEE Trans. Syst. Man Cybern.*, vol. 11, no. 2, pp. 109–125, 1981. DOI: 10.1109/TSMC.1981.4308636.

[9] P. Healy and N. S. Nikolov, "Hierarchical drawing algorithms", in *Handbook of Graph Drawing and Visualization*, R. Tamassia, Ed., Chapman and Hall/CRC, 2013, pp. 409–453.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman, 1979, ISBN: 0-7167-1044-7.

[11] B. Berger and P. W. Shor, "Approximation algorithms for the maximum acyclic subgraph problem", in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990, pp. 236–243.

[12] P. Eades, X. Lin, and W. F. Smyth, "A fast and effective heuristic for the feedback arc set problem", *Inf. Process. Lett.*, vol. 47, no. 6, pp. 319–323, 1993. DOI: 10.1016/0020-0190(93)90079-O.

[13] E. G. Coffman and R. L. Graham, "Optimal scheduling for two-processor systems", *Acta Informatica*, vol. 1, pp. 200–213, 1972. DOI: 10.1007/BF00288685.

[14] M. R. Garey and D. S. Johnson, "Crossing number is NP-complete", *SIAM Journal on Algebraic Discrete Methods*, vol. 4, no. 3, pp. 312–316, 1983. DOI: 10.1137/0604033.

[15] P. Eades and N. C. Wormald, "Edge crossings in drawings of bipartite graphs", *Algorithmica*, vol. 11, no. 4, pp. 379–403, 1994. DOI: 10.1007/BF01187020.

[16] U. Brandes and B. Köpf, "Fast and simple horizontal coordinate assignment", in *Graph Drawing, GD 2001*, P. Mutzel, M. Jünger, and S. Leipert, Eds., ser. Lecture Notes in Computer Science, vol. 2265, Springer, 2002, pp. 31–44. DOI: 10.1007/3-540-45848-4_3.

[17] M. Jünger and P. Mutzel, "2-layer straightline srossing minimization: Performance of exact and heuristic algorithms", *Journal of Graph Algorithms and Applications*, vol. 1, no. 1, pp. 1–25, 1997. DOI: 10.7155/jgaa.00001.

[18] G. Sander, "A fast heuristic for hierarchical manhattan layout", in *Graph Drawing, GD 1995*, F. J. Brandenburg, Ed., ser. Lecture Notes in Computer Science, vol. 1027, Springer, 1996, pp. 447–458. DOI: 10.1007/BFb0021828.

[19] C. Bachmaier, A. Gleißner, and A. Hofmeier, "Dagmar: Library for dags", Department of Informatics and Mathematics, University of Passau, Tech. Rep. MIP-1202, 2012.

[20] S. G. Kobourov, S. Pupyrev, and B. Saket, "Are crossings important for drawing large graphs?", in *Graph Drawing, GD 2014*, C. A. Duncan and A. Symvonis, Eds., ser. Lecture Notes in Computer Science, vol. 8871, Springer, 2014, pp. 234–245. DOI: 10.1007/978-3-662-45803-7_20.

A Files in the thesis archive

The thesis archive in the Information System of Masaryk University contains the following:

- `include/` - The code for the library. For the description of the contents of individual files see appendix B and for instructions on how to compile the library see appendix C.
- `example/` - Examples of the usage of the library. For build instructions see appendix C. Contains two subdirectories:
  - `draw/` - The code for the application described in section 3.2.
  - `simple/` - A simple demonstration of how to draw a manually created graph.
- `stats/` - The code used for collecting the data in section 4.2. It also includes the code for inserting cycles to graphs.
- `data/` - The graphs used for the comparison with *dot*. The subdirectories `20/`, `50/`, `100/` and `150/` contain the graph sets generated by DAGmar with inserted cycles. The directory `examples/` includes the graphs from section 4.1.
- `CMakeLists.txt` - A cmake file for building the example code.
- `LICENSE.txt` - The license.

B Source code structure

The source code for the library is contained in the `include/` directory in the thesis archive. It contains the following files:

- `interface.hpp` - This file should be included to use the library. It includes three headers `graph.hpp`, `layout.hpp` and `types.hpp` which provide the interface of the library.
- `graph.hpp` - The representation of a graph.
- `layout.hpp` - Contains the `sugiyma_layout` class representing the graph layout.
- `types.hpp` - The types which are part of the public interface.
- `subgraph.hpp` - Contains a representation of a subgraph used to represent the connected components of the input graph together with the code for breaking a graph into connected components. Additionally, it provides a data structure for mapping vertices to arbitrary data.
- `cycle.hpp` - The algorithm for cycle removal.
- `layering.hpp` - Provides a representation of a hierarchy and the code for the network simplex algorithm and for transforming a layering into a proper layering.
- `crossing.hpp` - The code for performing crossing reduction and counting crossings.
- `poositioning.hpp` - The positioning algorithm.
- `routing.hpp` - The algorithm for routing edges around nodes.
- `vec2.hpp` - The representation of a 2D vector together with the required vector operations.
- `utils.hpp` - Various data types and functions needed throughout the library.

- `report.hpp` - Contains a namespace with global variables meant for storing different diagnostics about the implemented algorithms for debugging purposes. However, the namespace is only available if the macro `REPORTING` is defined.

C Build instructions

To build a project which uses the library a C++ compiler supporting the C++17 standard is needed. Additionally, to build the examples and the code for collecting statistics the *cmake*[1] build system with a minimum version of 3.5.0 is required.

C.1 Using the library

To use the library in a project the only thing that is required is the `include/` directory contained in the thesis archive. It can then be used by simply including the file `interface.hpp`:

```
#include "path-to-include-directory/interface.hpp"
```

Assuming the only source file in the project is `main.cpp` it can be compiled by any compiler with C++17 support. On a UNIX system with clang it can be compiled using the command:

```
$ clang++ -std=c++17 main.cpp -o main
```

Alternatively, the file can be included as:

```
#include "interface.hpp"
```

And then compiled using:

```
$ clang++ -std=c++17 -I"path-to-include-directory" \
    main.cpp -o main
```

Yet another alternative is to use cmake and add the following lines for any target which requires the library to your `CMakeLists.txt` file:

```
target_compile_features(my_target PRIVATE cxx_std_17)
target_include_directories(
    my_target PRIVATE path-to-include-directory/
)
```

1. `https://cmake.org/`

C.2 Building the examples

The included `CMakeLists.txt` can be used to build the example applications. It provides four targets:

- `draw` - The application described in section 3.2.
- `simple` - A simple example showcasing how the library can be used to draw a graph embedded in the source code.
- `stat` - Collects statistics on all graphs in a given directory.
- `cycl` - Adds cycles to a graph described by a file in the DOT format.

On a UNIX system, assuming the content of the thesis archive was extracted into the current directory, the targets can be built by the following sequence of commands:

```
$ mkdir build && cd build
$ cmake ..
```

Then all of the targets can be built using:

```
$ make
```

Or just a single target:

```
$ make target
```

The usage of the resulting executables can by displayed by passing them the `-h` option.