Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Scalable Drawing of Nested Directed Acyclic Graphs With Gates and Ports

Master Thesis

T. Leu

August 20, 2021

Advisors: Prof. Dr. T. Hoefler, Dr. T. Ben-Nun

Department of Computer Science, ETH Zürich

Abstract

Graphs are a popular concept throughout all of computer science, from networking through data management to machine learning. This has given rise to the problem of automatic graph drawing, yet efficient and aesthetically-pleasing graph layouting is as diverse as the underlying graph characteristics. In this thesis we deal with graphs that not only can be nested in two different ways, but in which nodes also have ports. We derive two alternative methods from state-of-the-art approaches. Our focus is on both the quality of the drawings and the performance of the layouters. We then evaluate and compare our algorithms on a total of 40 real-world graphs. We find that at least one of them, a level-based algorithm, is able to generate adequate drawings in a scalable manner.

Acknowledgements

I am very grateful to my supervisor, Dr. Tal Ben-Nun for his valuable and practical advice as well as his great encouragement during the thesis. I wish to apologize for overrunning our weekly meetings on a regular basis. I also want to thank Dr. Prof. Torsten Hoefler for enabling and overseeing this thesis, as he did with my Bachelor Thesis.

A special thanks goes to my parents for supporting me both financially and emotionally during the entire study time; and to my friends and my fiancée Sarah for always having an open ear.

Contents

Chapter 1

Introduction

Ben-Nun et al. [7] have recently introduced the *Stateful DataFlow multiGraph* (SDFG) as an intermediate representation of a program in the context of high-performance computing (HPC). Therein, so-called *performance engineers* can interactively apply transformations to the SDFG that do not change the output of the program, but the dataflow within the program and thus improve performance. It is essential that the performance engineers are presented with a comprehensible, visual representation of the SDFG they are working on. This thesis therefore deals with the drawing of SDFGs.

While the study of automatic graph drawing has a rich and ongoing history, we believe that some of the features in the drawing of SDFGs – or at least their combination – are rather unique. We try to abstract those features in a new class of graphs that we call *nested directed acyclic graphs with gates and ports* (*NGP-DAG*). On one hand, such an abstraction helps us to capture the fundamental challenges of drawing SDFGs; on the other, it may be used in the future for other types of graphs with similar features and layout requirements.

We assess the quality of the generated drawings through a set of constraints (e.g., nodes must not overlap each other), and through a self-defined cost function that assigns to each drawing a score that reflects its comprehensibility. Further, for use in an interactive environment, it is important that our algorithms are fast. Other than algorithmic changes, we try to increase performance by employing a lower-level language (*WebAssembly*) and by multithreading.

We review two existing, fundamentally different approaches to generating layouts: A *level-based* approach (*Sugiyama framework* [68]) and a *force-directed* approach (*magnetic spring model* [67]). We adapt [1] these approaches to the NGP-DAG layout problem and suggest mechanisms that may improve the

[1]Our code can be found on https://github.com/thleu-ethz-ch/ngp-dag-layout.

results. One of those mechanisms is a global ranking scheme, which leads to edges overlapping nodes. To mitigate this issue, we introduce an algorithm that guarantees to resolve all overlaps. We also present a novel linear-time algorithm for mapping nodes to levels in a compact way.

We evaluate the quality and performance of our solutions on a number of real-world SDFGs. By the lack of a reference implementation we can only compare our own algorithms.

The thesis is structured as follows: In Chapter 2 we give a brief introduction to the SDFG and introduce the two approaches *level-based* and *force-directed* as well as their respective realizations *Sugiyama framework* and *magnetic spring model*. Before we adapt these algorithms, we define the NGP-DAG model and show how to map an SDFG to it in Chapter 3. In Chapter 4 we discuss how a drawing should look and define the cost function as a metric of how well a given drawing matches our goals. The following Chapter 5 and Chapter 6 explain how we adapt and augment the Sugiyama framework and the magnetic spring model, respectively. We then discuss how we use WebAssembly and multithreading to boost performance in Chapter 7. After that, we evaluate and compare our various ideas in Chapter 8. The thesis closes with our conclusions and notes on potential future work (Chapter 9).

Chapter 2

Background & Related Work

In this chapter we lay out the background on which we base our work.

2.1 Notation

A directed graph G is a pair (V, E) where V is a set of nodes and E a set of edges. An edge e is a pair $(u, v) \in V \times V$. We call u the *origin* and v the *destination* of e. We define the *in-neighbors* of a node v as $N^-(v) := \{u \in V : (u, v) \in E\}$, and the *out-neighbors* as $N^+(v) := \{w \in V : (v, w) \in E\}$. The union of in- and out-neighbors of v is the set of *neighbors* $N(v)$.

2.2 Stateful DataFlow multiGraph

The *Stateful DataFlow multiGraph* (SDFG) is an intermediate representation of a computer program. As such it is detached from the programming language and even the targeted hardware. *Stateful dataflow* combines the concepts of dataflow [73] and traditional control-centric program definitions. In the following we outline some of the elements of the SDFG that are relevant to layouting. For a complete description of all the elements that may appear in an SDFG and their semantics, we refer the reader to the original paper [7].

Figure 2.1 showcases a complete, valid SDFG. On the outside the SDFG is a state machine and on the inside each state is a dataflow acyclic multigraph. It is also perfectly possible and not uncommon to have programs whose execution order is only determined by dataflow. In this case the state machine has just one – the initial – state.
Nodes of the dataflow diagram can be either data containers (e.g., *access nodes*), computations (e.g., *tasklets*) or special constructs for concurrency (e.g., *maps*); edges represent data movement. To distinguish between different streams of data, nodes may be equipped with *connectors*, where edges may

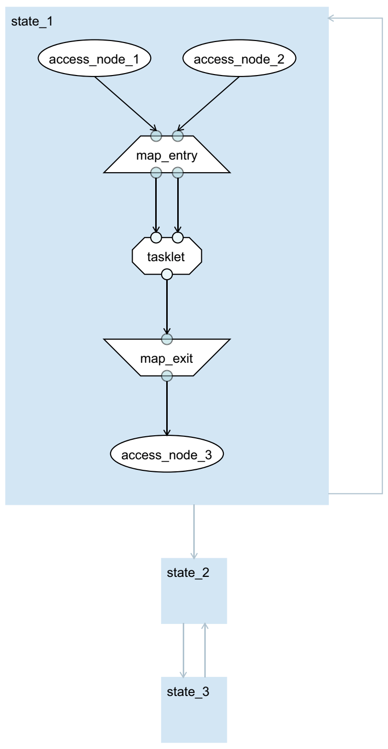


Figure 2.1: Example of an SDFG with three states. The states and the edges between them are drawn in light blue. As shown, states may be empty. The first state contains a dataflow diagram consisting of three access nodes, a map and a tasklet. Dataflow nodes are depicted with white background and a distinct contour for each type of node. The translucent circles at the top and bottom of the map entry and exit nodes are *tunnel connectors*; they form pairs (*tunnels*) that are always vertically aligned. The opaque circles attached to the tasklet are *simple connectors*.

start from (*out-connector*) or end at (*in-connector*). A map marks a parallel scope that is delimited by a *map entry* and a *map exit* node. These boundary nodes can have *tunnel connectors*, pairs of connectors – one in-connector and one corresponding out-connector – that allow data flow from the outside of the scope to the inside and vice versa. They act like a tunnel that connects the edge ending at the in-connector and the edge starting at the out-connector. In contrast, *simple connectors* are individual and do not have a complementary connector.
SDFGs can be further nested. A *Nested SDFG node* is a computation node similar to the tasklet, but consists of an entire, self-contained SDFG.

2.3 Automatic Graph Drawing

Graphs are a simple yet powerful tool to model any relation appearing in the real world. This has naturally stirred the desire to generate drawings of such graphs automatically instead of drawing them by hand (which is virtually impossible with the sizes of some graphs today). The topic of automatic graph drawing is multifaceted and a variety of algorithms have been developed to match the diverse characteristics of graphs. A good overview can be found in the 'Handbook of Graph Drawing and Visualization' [69]. Before we look at some of the existing algorithms, we want to address the question of what a well-drawn graph looks like.

2.3.1 Aesthetics

Despite the subjective nature of this question, a number of objective criteria have come forth, e.g., the number of edge crossings, the number of edge bends, the orthogonality of edges and the sum of the edge lengths. While such metrics have traditionally been based upon the authors intuition [6, 57], more recent work has also taken into account insights from psychology and human perception. For example, Ware et al. [75] derived metrics from the well-known Gestalt laws [47] and Van Ham and Rogowitz [72] asked users to draw a graph themselves, in order to reverse-engineer aesthetic criteria they perceive important.

Apart from generally looking visually pleasing, a good drawing is one that can be interpreted correctly and promptly by the target audience. Various user studies were conducted in order to empirically measure the effect of certain aesthetic criteria on the interpretability of a drawing [58, 56, 59, 40, 75]. The surveys in general agree that the single most outstanding differentiator for interpretability is the number of **edge crossings** present in the drawings. Other empirically significant metrics include:

- **Edge bends**: the number of intermediate points on edges where the edge changes direction [58]
- **Path bends**: similar to the edge bends, but also taking into account changes of direction at nodes [75]
- **Edge lengths**: the total geometric line length of the edges [75]
- **Crossing angles**: if there are crossings, how close to 90 degrees the crossing lines are [40]

2.3.2 Drawing Algorithms

Following the diversity of relations to be modeled with graphs, a variety of drawing styles and appropriate algorithms have been developed. Some classes of graphs allow for particularly aesthetic representations with regard to the aforementioned criteria. For example, planar graphs can be drawn with zero crossings [13, 52], and biconnected 4-graphs [1] can be realized as

[1]A biconnected graph is one that is still connected after removing an arbitrary node. A 4-graph is one with maximum degree 4.

orthogonal drawing [9], that is, all crossing angles are 90 degrees. Another orthogonal drawing approach allows higher degree graphs, but requires them to be planar again [24]. While a graph can be transformed into a planar graph (*planarization*) by replacing every edge crossing with a dummy node (to be removed later such that in its place is just an edge crossing again) [5], this does not necessarily result in a graph with a small number of crossings. In the next two sections we will look at techniques that work on a large range of graphs without any requirements regarding planarity, maximum degree or connectivity. For ease of discussion we will however assume that the graphs are weakly connected. Note that whenever a graph is not connected, we can find its connected components in linear time [39] and lay the components out separately.

2.3.3 Hierarchical Graph Drawings [2]

One broad class of graphs that frequently arise in practice are *Directed Acyclic Graphs* (DAGs). Every DAG encodes a partial order that can be interpreted as a hierarchy. The graph drawing community has therefore come up with ways to draw DAGs such that those hierarchies are easily recognizable. Two important classes of layouts have emerged: *radial drawings* [17] where nodes lie on concentric circles, and *upward drawings* where for all edges the y-coordinate of the head is larger than that of the tail. Note that, counterintuitively, an upward drawing on a computer screen has edges that point downward, because the y-axis is also pointing downward.

Sugiyama Framework

In this regime, a framework based on the work of Warfield [76] in the context of *Interpretive Structural Modelling* (ISM) [61] and refined by Sugiyama et al. [68] (therefore known as *Sugiyama framework*) and Gansner et al. [29], has become a standard technique for layouting hierarchical graphs. To this day, it is typically the basis for one of several available layout algorithms in general purpose graph drawing software, including *Graphviz* [30] (under the name `dot` [29]), the *Microsoft Automatic Graph Layout* (MSAGL) [53] and Mathematica [41]. The JavaScript library `Dagre` [55] that we are going to use in a baseline implementation is also in large part adapted from Gansner et al. [29].

While being particularly well-suited for DAGs, the Sugiyama framework can also be used for directed graphs with cycles or even for undirected graphs. All that needs be done is (re-)orienting edges such that there are no cycles, for example, by starting a depth-first-search (DFS) from an arbitrary node

[2]The terms *hierarchical graph* or *hierarchical drawing* must not be confused with the graph nesting that arises in SDFGs and also constitutes a hierarchy.

and orienting the edges in the direction of their traversal. After placing the nodes, the original orientations can be restored. Since every reversed edge will ultimately be drawn against the general flow, some effort was made to choose a minimal set of edges whose reversal makes the graph acyclic. This so-called *minimum feedback arc set* problem is NP-hard [35], but there are approximations that run in polynomial time [8] or linear time if $|E| \in \mathcal{O}(|V|)$ [20].

The Sugiyama framework works by restricting nodes to lie on horizontal lines, called *ranks* (sometimes also *levels* or *layers*) such that edges point strictly downwards. Assuming cycles have been removed in a preprocessing step as described in the previous paragraph and we have a now a DAG, the framework consists of the following steps:

1. **Rank Assignment**: Partition the nodes into ranks such that all in-neighbors of a node have a lower rank than the node itself ($\forall (u, v) \in E : rank(u) < rank(v)$). The ranks are labeled from top to bottom and thus, all edges point downwards. Edges may span multiple ranks [3].
2. **Inserting Dummy Nodes**: Create a *proper hierarchy* [76]. That is, edges are modified to span exactly one rank, by placing a *dummy node* (sometimes called *virtual node*) on each intermediate rank and rerouting the edge through those dummy nodes.
3. **Permuting Nodes Within Ranks**: Permute the nodes within each rank with the aim of reducing the number of edge crossings.
4. **Node Coordinate Assignment**: Place the nodes in the plane, that is, assign x- and y-coordinates to them.
5. **Edge Coordinate Assignment**: Restore the original edges and let them pass through the corresponding dummy nodes.

An attractive feature of this framework is its modularity. For every nontrivial step (1, 3, 4) one can choose between multiple viable alternatives that already exist or readily invent a new one without worrying about the other steps. In the following, we describe some of the existing options, without any claim to completeness.

Step 1: Assigning Ranks

As outlined above, the primary goal of the ranking step is to have all edges point downwards. Probably the simplest algorithm to achieve this is the *longest path* algorithm [51]. It puts all sinks in the lowest rank and then gradually adds nodes from bottom to top where each node is placed as low as possible, and only after all out-neighbors have already been placed. That

[3]We say an edge (u, v) spans $(rank(v) - rank(u))$ ranks.

is, the order in which nodes are placed is given by a reverse topological sort. Besides its simplicity and low complexity of $\mathcal{O}(|E|)$, the longest path algorithm has the advantage that it always produces a ranking with the minimum number of ranks (equal to the length of the longest path between any pair of nodes plus one), or put differently, it generates drawings of minimal height. However, it has been criticized [21, 36] that the drawings might be too wide, in particular in the lowest rank where all the sinks are.

Gansner et al. [29] proposed to minimize the sum of the edge spans. Besides the direct positive effect on the aesthetic goal of having short edges (see Section 2.3.1), this reduces the number of dummy nodes that have to be added in Step 2 and hence decreases the runtime of subsequent steps. They formulate an integer linear program (ILP) with $|V|$ variables and $|E|$ constraints, where the constraint matrix is unimodular. While a standard LP solver can solve this in polynomial time [50], they also provide a solution in the form of a *network simplex* algorithm [15]. They do not give any bound on the complexity of this algorithm [4], but claim that in practice the algorithm takes only few iterations.
It is also worth noting that in the initialization of their network simplex algorithm, they create a valid ranking which is already compact in the sense that each node is connected to the rest of the graph through a *tight* edge [5]. Hence this initialization alone constitutes another legitimate candidate for a ranking algorithm. The rankings thereof may be better than those of the longest path algorithm in terms of the number of dummy nodes and tendentiously also in terms of total width – because sinks are not restricted to lie in the bottom rank. The time complexity of their initial ranking is in $\Omega(|E| \log |E|)$. We introduce in Section 5.3 a different algorithm that achieves the same in linear time.

An improvement over the approach of Gansner et al. (and other approaches) was introduced and studied by Healy and Nikolov [36, 37]. In addition to minimizing the edge spans – or equivalently the number of dummy nodes – they let the user impose constraints on the maximum admissible width and height. They call this extended problem *WHS-Layering* (Width, Height, Spans) and point out that it is NP-hard. Nevertheless they describe a branch-and-cut algorithm that solves the problem exactly and report that it can be applied to graphs with up to 100 nodes in reasonable time.

[4]By now, due to Tarjan [70] a version of the network simplex algorithm is known which runs in $\mathcal{O}(|V||E| \log |V| \log(|V|C))$ in the worst case where C is the maximum cost of an edge.

[5]We formally define tight edges in Section 5.3. Informally, a tight edge is one that is not unnecessarily long.

Step 2: Adding Dummy Nodes

This is a trivial step and can be performed by looping through all edges and replacing any edge that spans more than one rank with a path through newly inserted dummy nodes (one on each intermediate rank). The complexity of this step is $\mathcal{O}(|E| + |D|)$ where D is the set of all dummy nodes at the end of step 2. Note that for some graphs, $|D|$ is at least $\Omega(|E|^2)$, regardless of the ranking algorithm. For example, take a graph whose edges are divided into one long path of length $\frac{|E|}{2}$ and $\frac{|E|}{4}$ shortcuts with 2 hops each between the first and last node of this path. To properly layer this graph, one has to insert at least $\frac{|E|}{4}(\frac{|E|}{2} - 2)$ dummy nodes [6].

We denote by V' the set of nodes at the end of Step 2. That is, V' contains both the original nodes E and the dummy nodes D. Similarly, we call the new set of edges at the end of this step E'. Observe that $|E|' \in \mathcal{O}(|E| + |D|)$.

Step 3: Permuting Nodes Within Ranks

At this point we have a proper hierarchy and if we were just interested in creating a valid, but aesthetically suboptimal upward drawing we could skip this step. However, recall that minimizing the number of crossings is considered critical to the comprehensibility of a graph. To this end, the nodes can be rearranged within their ranks. For a graph with just two ranks, the problem – called *Two-Layer Crossing Minimization* – is NP-hard [31]. Even after fixing the order in one of the ranks (*One-Sided Crossing Minimization* or *OLCM*) it is still NP-hard [22].

The more general problem, where we have k ranks and none of them is fixed, is much more difficult to solve. Therefore only few attempts have been made to solve it directly [42, 34]. The usual way to approach the problem is to (repeatedly) apply OLCM layer by layer, alternatingly from top to bottom and bottom to top [68, 29, 23].

In the previous paragraph we established that Step 3 is based on OLCM and in the first paragraph we noted that OLCM is NP-hard. Nevertheless, Jünger and Mutzel [43] suggested an exact branch-and-cut algorithm for solving OLCM exactly and state that it runs reasonably fast for up to 60 nodes per rank. In the same paper, they compare several heuristics for the problem through experiments and state that the *barycenter heuristic* (as used by Sugiyama et al. [68]) should be the heuristic of choice, outperforming others in both number of crossings and runtime [7]. The barycenter heuristic works as follows: For each node in the rank to be reordered, calculate the average position of its neighbors in the adjacent rank whose order is fixed.

[6]The upper bound for simple graphs can also be stated as $|D| \in \mathcal{O}(|V||E|)$ [23], but it does not hold for multigraphs.

[7]Though others have ascribed the median heuristic better results, see e.g. Barth et al. [4].

Then sort the nodes by those averages. The new order is only adopted if the number of crossings with the new order is smaller than the old one. This calls for an efficient way of counting crossings between two layers. Barth et al. [4] describe both an algorithm with $\mathcal{O}(|E'_r| + C_r)$ and one with $\mathcal{O}(|E'_r| \log(\min\{|V'_r|, |V'_{r+1}|\}))$ runtime where V'_r is the set of nodes in rank r, E'_r the set of edges between rank r and $r+1$, and C_r the number of crossings between those two. It turns out that the latter is typically faster in practice.

Step 4: Node Coordinate Assignment

As soon as the ranking is determined, the y-coordinates of the nodes can be obtained trivially. This is done by calculating the height of a rank as the maximum height of one of its nodes. Then the ranks are assigned y-coordinates, such that there is some fixed distance between two neighboring ranks. If nodes do not have uniform height, they can be vertically aligned at the top, middle or bottom of their rank.

The x-coordinates then must be assigned separately, respecting the node order determined in the previous step. Sugiyama et al. [68] and Gansner et al. [29] both make it a priority that paths formed by dummy nodes – called *inner segments* – are vertical. They formulate the problem of coordinate assignment as quadratic or linear program, respectively. In the case of Gansner et al. it comes down to minimizing the x components of all edges. They also give a solution in the form of another network simplex algorithm (see Step 1).

Several heuristics have been proposed of which we want to highlight just one. The algorithm introduced by Brandes and Köpf [11] runs in only $\mathcal{O}(|V'| + |E'|)$ time and guarantees vertical inner segments – as long as inner segments are not crossed by other edges. (But as they show, removing those crossings can easily be implemented as a preprocessing step). The idea behind their algorithm is simple: try to vertically align each node with one of its up- and one of its down-neighbors. By prohibiting any crossing of an inner segment and due to dummy nodes having exactly one up- and one down-neighbor, they are always aligned vertically.

Step 5: Edge Coordinate Assignment

This is again a trivial step. After all nodes have been placed, the original node and edge set have to be restored. The positions of the dummy nodes are then points that define the paths of the edges. The first and last point of an edge are determined by the position of its origin and destination nodes.

Alternatives to the Sugiyama Framework

An algorithm that may still be seen as a realization of the Sugiyama framework, but is different with regard to the dummy nodes, is given by Eiglsperger

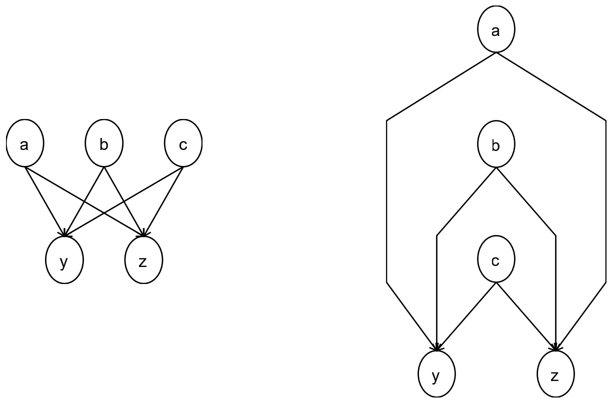


Figure 2.2: Two drawings of the complete bipartite graph $K_{3,2}$. On the left, a drawing with 2 ranks and 3 edge crossings. Any permutation of the nodes results in the same number of crossings. On the right, a drawing with 4 ranks and no crossings at all. This example shows that the aesthetic goals *edge length minimization* and *crossing minimization* are mutually exclusive.

et al. [23]. They introduce at most 2 dummy nodes per edge and thus obtain $|V'| \in \mathcal{O}(|V|)$ and $|E'| \in \mathcal{O}(|E|)$ which allows them to give better bounds on the runtime complexity. As a consequence, the problems of the subsequent steps have to be solved differently and standard algorithms may not be applied.

The disadvantage of breaking up the layouting problem into a sequence of simpler subproblems is that they are treated as if they were independent, which they are not. Knowing that they depend on each other does not directly help, because the framework does not allow any interaction between them except passing the output from one step as input to the next one. For example, generating a highly compact ranking in Step 1 might inevitably lead to more crossings in Step 3 compared to one with more ranks (see Figure 2.2 for an example). On the other hand, there is no point in discarding a compact ranking in favor of a looser ranking – that induces longer edges – without knowing whether the compact ranking would generate (significantly) more crossings in the subsequent step than the loose one.

This problem was tackled by Chimani et al. [14] who combined steps 1-3 in a way that prioritizes crossing minimization. Their algorithm based on a planarization approach (see also Mutzel [54]) generates layouts with substantially fewer crossings compared to the classic Sugiyama implementation. This comes at the cost of longer edges and higher running time. Utech et al. [71] have combined the same steps with a genetic algorithm where the genetic representation encodes both the rank assignment and the order of the nodes within ranks. They also report fewer crossings but increased running time. For the case where the input is not already a DAG, Rüegg et al. [60] have combined cycle removing and ranking.

Another well-known class of drawing algorithms are *force-directed algorithms*, which we outline in the next section.

2.3.4 Force-Directed Methods

As the name suggests, force-directed drawing algorithms annotate each configuration (position of the nodes) with certain forces. Typically, the nodes are viewed as objects that are affected by some physical forces such as the elastic force (spring force), the electrostatic force (Coulomb force) or magnetic fields. The goal is then to find a configuration where the net force of all nodes – sometimes called *energy* or *stress* – is minimal. There are essentially two approaches to this: *Integration*, where alternately the nodes are displaced in the direction of their net force and the forces for the new configuration are calculated [19, 26]; and *derivation*, where a minimum is sought by solving a system of equations [44, 12, 18]. While the integration approach is easier to implement and has controllable runtime – the number of iterations can be chosen freely – it is more likely to converge to one of many suboptimal local minima compared to the direct solution. A method to mitigate the local minima but following the iterative integration is constituted by the use of simulated annealing [16].

The idea to model nodes as rings and edges as springs goes back to Eades [19]. However, he did not use the physical law of ideal springs, Hooke's Law, for calculating the forces. Fruchterman and Reingold [26] changed this in their adaptation of the algorithm, and point out that neither Eades' nor their algorithm is a realistic simulation of a process that could happen in nature. The spring force acts as either repulsive or attractive force: When the spring is in stretched state, the rings are pulled towards each other; when it is in compressed state, they are being pushed away from each other. Additionally, towards a uniform distribution of the nodes, electrostatic forces can be employed to model repulsion between all pairs of nodes. Moreover, Kamada and Kawai [44] model spring forces between all pairs of nodes. They choose the natural length of each spring in relation to the graph theoretic distance of the nodes, that is, the length of the shortest path between them.

The algorithm of Kamada and Kawai is implicitly a form of *stress minimization*, a method first used in *multidimensional scaling* (MDS) [48] and later applied to graph drawing [49]. The *total stress* to be minimized is given by the residual sum of squares regarding the difference between the Euclidean distance and some theoretically ideal distance. A popular variant of stress minimization by Gansner et al. [28] uses *majorization* which is faster than the Newton-Raphson method used by Kamda and Kawai. Some recent work [78] suggests that *Stochastic Gradient Descent* (*SGD*) applied to stress minimization may converge faster and to lower stress levels more consistently than majorization.

When the main goal of a drawing is to capture the distance between nodes, MDS is undoubtedly the method of choice. Yet it is not clear, that MDS – or other force directed methods – are a good fit for visualizing directed graphs in general, and hierarchical graphs such as flow diagrams in particular. For example, none of the ideas presented in this section so far care about the continuity of flow – a Gestalt principle [47] that is addressed in the Sugiyama framework by forcing all edges to point downwards.
Few suggestions in this direction have been made within the family of force-directed methods: In *constrained majorization* as introduced by Dwyer et al. [18] it is possible to constrain the x- and y-coordinates of the edges to form an upward drawing. Carmel et al. [12] proposed to minimize an energy function separately for the x- and y-axis that favors edges pointing in the same direction. Sugiyama and Misue [67, 66] extended the spring model with an additional magnetic force. When the magnetic field is parallel, edges are attracted towards an upward drawing. We choose this magnetic spring model as candidate for a force-directed layout algorithm for its simplicity and extensibility.

All the aforementioned force-directed methods have a runtime complexity of $\Omega(|V|^2)$ and hence they might be too slow for graphs with more than 1,000 vertices. To counter this, various divisions of the problem into smaller problems have been found. The force-directed algorithm of Barnes and Hut [3] runs in $O(|V| \log |V|)$ by the use of a tree-structured hierarchical subdivision of space into quadratic cells. The multi-level approach of Gajer et al. [27] chooses a maximal independent set of vertices on each level and achieves a time complexity of $O(|V| \log^2 |V|)$. Another multi-level scheme, based on spectral partitioning and with complexity $O(|V|^{1.5} + |E|)$ is due to Frishman and Tal [25]. Despite its complexity, their algorithm is reportedly fast in practice because it is tailored to efficiently run on GPUs.

2.3.5 Ports

Recall from Section 2.2 that edges in the SDFG can be between connectors rather than just nodes. Within the Sugiyama framework, the possibility of edges that do not end at node centers but rather at distinct points of the nodes' borders – so-called *ports* – was already stated in the seminal paper of Gansner et al. [29]. However, they use this information only in the context of assigning x-coordinates to the nodes, to promote vertical edges in the presence of ports. Waddle [74] suggested to use the positions of the ports – assumed to be fixed – already in the preceding step of permuting the nodes to reduce the number of crossings. Schulze et al. [65] expanded on this idea and provide an algorithm that can reduce the number of crossings even further by reordering ports within nodes. We adopt this algorithm as described in Section 5.6.

Now that we have introduced the SDFG model and some important methods for automatic graph drawing, we will discuss the challenges specific to drawing SDFGs in the next chapter.

Chapter 3

Generalizing the SDFG Layout Problem

Recall from Chapter 2 that an SDFG is a nesting of two different types of graphs – the state graph and the dataflow graph. The state graph is a directed graph that may contain loops and other cycles; the dataflow graph is a directed acyclic (multi-)graph (DAG). In the interest of a unified algorithm for both types of graphs, we first transform the state graph to a DAG. We propose to draw loops on the left or right side of a node, from bottom to top (see Figure 2.1). This makes them easily recognizable and by reserving a bit of extra width inside the node for drawing it, the loop can not cross any other edges or overlap any nodes. Other cycles can be temporarily removed by reorienting some of the edges. After the layout is computed, the original orientations are restored. We describe an algorithm for choosing the edges to be reversed in Section 3.1.

Although in the original SDFG definition [7] maps are defined only implicitly through a map entry and a map exit, we find it desirable that the separation of the inside and outside of a map is visually apparent. To this end, we model a map as its own node with the contents of the map scope nested inside. Maps differ from other nodes with children (namely states and nested SDFG nodes) in having an entry and an exit node that connect the inside with the outside through tunnel connectors. The entry and exit node are conceptually part of both the parent and the child graph. This is illustrated in Figure 3.1. To capture this principle, we distinguish between two types of child graphs, *isolated* and *gated* ones. States and nested SDFG nodes are allocated isolated child graphs, whereas maps are allocated gated child graphs with the map entry and exit nodes as gates.

An isolated child graph can consist of multiple connected components. If this is the case, they are laid out separately and then placed side by side. Note that identifying the components takes $\mathcal{O}(|V| + |E|)$ time [39].

The connectors can be modeled as ports. Tunnel connectors, that come in

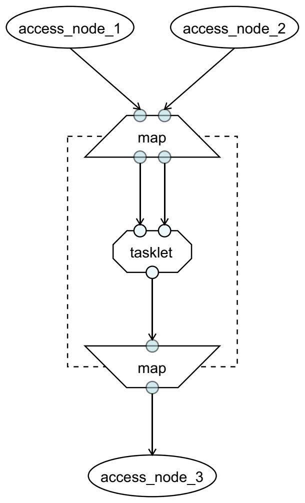


Figure 3.1: Conceptual drawing of a map. The dashed rectangle marks the boundary of the map node. The map entry and exit nodes act as gates between the outside and the inside.

pairs, have to be vertically aligned for clarity. Simple connectors can be placed arbitrarily on the top (in-connectors) or bottom (out-connectors) of the node. We convey these layout requirements regarding connectors to ports. While in the SDFG not all edges are between connectors, we add dummy ports where they are not.

3.1 Removing Cycles

Observe that a graph with cycles can be made acyclic by reorienting some of its edges. It suffices to impose a total order on the nodes and then orient each edge such that its origin is before its target in this ordering. This is used in our algorithm, where the total order corresponds to the order in which nodes are visited. The graph is assumed to be connected.
Algorithm 1 starts with a topological sort [46] (‘toposort’) from an arbitrary node. If not all nodes are visited, there are cycles present in the graph. In that case, one of the unprocessed nodes that is not a sink is chosen as the next node. After reorienting its edges, it is the starting node for another toposort. This is repeated until all nodes are visited. Algorithm 1 runs in linear time (see Lemma 3.1) and guarantees to keep sources and sinks intact (meaning they are still sources and sinks, respectively, after applying the algorithm).

Lemma 3.1 *Algorithm 1 terminates in $O(|V| + |E|)$ steps.*

Proof If $|N^-(v)|$ and $|N^+(v)|$ are known for all nodes $v \in V$, the initialization takes $\mathcal{O}(|V|)$ steps. If not, they can be obtained in $\mathcal{O}(|E|)$ steps. Initially, both q and $q2$ are empty. Once they become empty again, both the main loop (line 8) and the inner loops (line 9 and 19) terminate. Hence it suffices to show

Algorithm 1 Removing Cycles

```
1:  $q.init(); q2.init()$                                  ▷ initialization
2:  for all $v \in V$ do
3:      $done[v] \leftarrow false$
4:      $pred[v] \leftarrow |N^-(v)|$
5:      $succ[v] \leftarrow |N^+(v)|$
6:      if $pred[v] = 0$ then
7:          $q.push(v)$
8:  repeat
9:      while not $q.empty()$ do                               ▷ toposort
10:         $u \leftarrow q.pop()$
11:         $done[u] \leftarrow true$
12:         for all $v \in N^+(u)$ do
13:             $pred[v] \leftarrow pred[v] - 1$
14:             if $pred[v] = 0$ then
15:                 $q.push(v)$
16:             else
17:                 $q2.push(v)$
18:     if not $q2.empty()$ then
19:         repeat                                       ▷ find next node $w$
20:             $w \leftarrow q2.pop()$
21:         until (not $done[w]$ and $succ[w] > 0$) or $q2.empty()$
22:         if not $done[w]$ and $succ[w] > 0$ then
23:             for all $v \in N^-(w)$ do            ▷ turn edges away from $w$
24:                 if not $done[v]$ then
25:                     $reverseEdge(v, w)$
26:                     $pred[v] \leftarrow pred[v] + 1$
27:             $q.push(w)$
28: until $q.empty()$ and $q2.empty()$
```

that the number of elements added to these queues is in $O(|V| + |E|)$. To q, every node is added exactly once, namely when the number of predecessors becomes zero – giving us $|V|$ insertions. The only insertion into $q2$ is in line 17. Considering that u is unique in line 10, we can associate each execution of line 17 with a unique edge (u, v). Thus we add at most $O(|E|)$ elements to $q2$. □

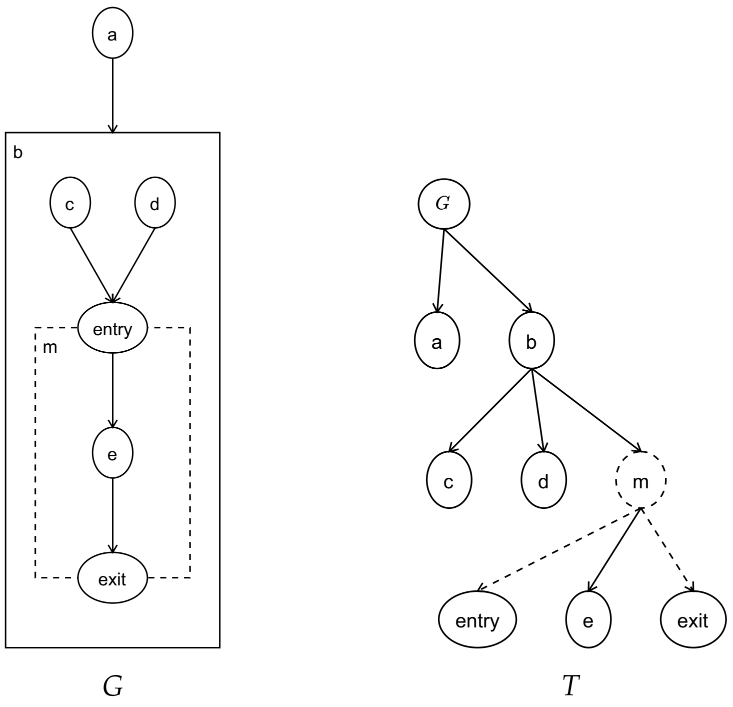


Figure 3.2: Example of a nested graph (G, T) with two child graphs: b is an *isolated* child graph and m a *gated* one. On the right, dashed edges mark the relation between the gated child graph and its designated *entry* and *exit*. The root node in T corresponds to the entire graph G.

3.2 Nested Directed Acyclic Graphs with Gates and Ports

We define a *nested* graph as a pair (G, T) where the nodes of $G = (V, E)$ are additionally arranged in a directed rooted tree $T = (V, E_T)$ and it holds that $\forall(u,v) \in E : \exists p \in V : \{u,v\} \subseteq children(p)$, where $children(u) := \{v \in V : (u,v) \in E_T\}$ are the children of node u in T. That is, edges in G are only allowed between siblings in T.

We call the subgraph induced by the children of a node u the *child graph* of u, $G_{child}(u) := G[children(u)]$. By restricting the edges to be between nodes of the same child graph, all child graphs are *isolated*. We promptly define a second type of child graphs: A *gated* child graph is similar to the isolated one, but with the addition that there is a distinct *entry* node $entry(G_{child}(u)) \in children(u)$ and a distinct *exit* node $exit(G_{child}(u)) \in children(u)$ with $entry(G_{child}(u)) \neq exit(G_{child}(u))$. We call the entry and exit of a gated child graph its *gates*. The condition on the edges is then relaxed such that gates may have edges to the outside: $\forall(u,v) \in E : \exists p \in V : u \in \bigcup_{w \in children(p)} \{w, exit(G_{child}(w))\} \wedge v \in \bigcup_{w \in children(p)} \{w, entry(G_{child}(w))\}$. See Figure 3.2 for an example of a nested graph with a gate.

In a graph with *ports*, each node $v \in V$ is associated with a set of *in-ports* $P_{in}(v)$ and a set of *out-ports* $P_{out}(v)$. The edges are then not just between nodes, but from an out-port to an in-port. That is, we have a function $p : E \to P_{out} \times P_{in}$ that maps an edge (u, v) to a pair of ports (p_u, p_v) such that $p_u \in P_{out}(u)$ and $p_v \in P_{in}(v)$. In order to allow flow passing through them, gates may be equipped with *tunnels*. A tunnel of a gate v is a pair $t := (t_{in}, t_{out})$ where $t_{in} \in P_{in}(v)$ and $t_{out} \in P_{out}(v)$.

The combination of these concepts gives raise to the *Nested Directed Acyclic Graph with Gates and Ports* that we will abbreviate with NGP-DAG in the following.

3.3 The NGP-DAG Layout Problem

Like in the general graph layout (or graph drawing) problem, the NGP-DAG layout problem consists of positioning nodes and edges. That is, we ascribe each node $v \in V$ a rectangle defined by the x- and y-coordinate of the top-left corner, a width and a height; and to each edge a straight line path, that is described by a list of points such that the first point touches the origin and the last point the destination. To make the hierarchy clear, we require the edges to point downwards, that is, the y-coordinate of the last point must be larger than the y-coordinate of the first point. We want to avoid nodes overlapping each other, and edges overlapping nodes. In the presence of ports, we also prefer to position those in a non-overlapping way. We restrict in-ports to lie at the top of a node, and out-ports at the bottom. Tunnels have to be vertical, that is, the corresponding in- and out-port must share the same x-coordinate. To reflect the nesting of the graph, we expect child graphs – including nodes and edges – to lie completely inside the rectangle assigned to their parent. Hence the width and height of a parent node have to be adapted to the size of its child graph. The node sizes can thus not be given as input; instead, we take as input the minimum sizes of the nodes. That is, there are functions $w_{min} : V \to \mathbb{R}_+$ and $h_{min} : V \to \mathbb{R}_+$ that assign each node a minimum width and a minimum height, respectively. The layout algorithm can then only enlarge nodes but not shrink them.

Chapter 4

Our Take on Aesthetics

In the previous section we defined some basic constraints on the layout. However, they are not sufficient to create visually pleasing and comprehensible drawings. We therefore introduce a *cost function* that tries to capture how hard it is to read a given layout of a graph. Purchase [57] suggested a relative cost function that returns values between 0 and 1, but we find that the upper bounds used therein are too lax to make for a meaningful cost function. Hence, we propose an absolute cost function. It will inherently assign larger costs to larger graphs but this can be seen as in line with the fact that larger graphs are also inherently harder to read. Our cost function is a weighted combination of edge crossings, edge bends and edge lengths.

Edge Crossings There is a consensus that the primary hindrance in the comprehensibility of a drawing is given by crossed edges. It has further been found that crossings at a right angle or close to it are less impeding than very small angles. We therefore penalize each crossing depending on the crossing angle. More precisely, we define the cost of a segment crossing as

$$cost(crossing) := 1 + cost(angle) = 1 + \frac{\cos(2 \cdot angle) + 1}{2}$$

where *angle* is any of the four angles between the segments. Taking any of them is possible because of the symmetry in the function (see Figure 4.1). Observe that this induces a cost between 1 and 2 for each crossing. The total crossing cost $cost_c$ then is

$$cost_c = \sum_{crossing} cost(crossing)\,.$$

Edge Bends The continuity of flow is another important characteristic of a readable graph. Though this is already reflected by the fact that we constrain

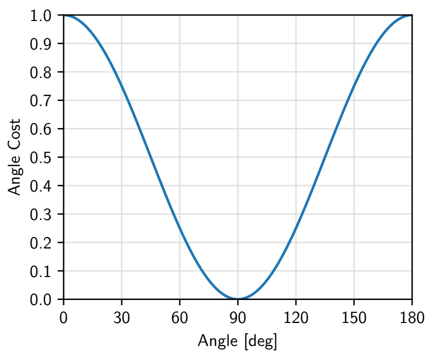


Figure 4.1: We define the angle cost of a crossing as $cost(angle) = \frac{\cos(2 \cdot angle)+1}{2}$.

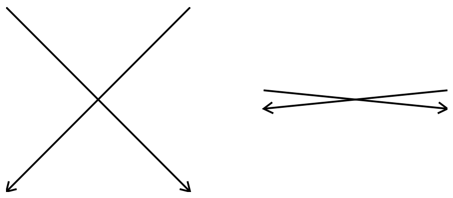


Figure 4.2: The angle cost for the two crossings are 0 and ≈ 0.96, respectively.

ourselves to an upward drawing where possible, we additionally want the edges to be as straight as possible. We simply define the total 'bendiness' cost [57] $cost_b$ as the total number of bends:

$$cost_b := \sum_{e \in E} (|points(e)| - 2)$$

where $points(e)$ is the list of points assigned to edge e.

Edge Lengths Finally, short edges are preferable over long edges whenever possible. Assuming the ideal edge length is given as parameter d to the layout algorithm, we define the length cost as the relative deviation of the actual edge length (sum of the lengths of its segments) $length(e)$ from d. The total length cost $cost_l$ is thus

$$cost_l := \sum_{e \in E} \frac{|length(e) - d|}{d} .$$

The three aforementioned cost factors can then be combined into an overall cost function for a generated layout:

$$cost(layout) := w_c \cdot c_c + w_b \cdot c_b + w_l \cdot c_l .$$

Choosing the weights $w_c = 1$, $w_b = 0.2$ and $w_l = 0.1$ yields ratings that we personally deem aesthetically adequate.

In the next two chapters we develop algorithms that aim to produce layouts of low cost, and in short time.

Chapter 5

Level-Based Approach

In this chapter we discuss our first approach to generating NGP-DAG layouts, which follows the renowned Sugiyama framework [76, 68, 29].

5.1 `Dagre`-Based Algorithm

`Dagre` [55] is a JavaScript library that implements Sugiyama-style drawing. Our initial solution is therefore based on this library and constitutes a baseline for the more refined solutions.

Recall that the size of a node depends on the layout of its child graph. Therefore it makes sense to lay the child graphs out recursively, starting at the innermost child graphs – the leaves of T [1].

`Dagre` does not offer support for ports. Thus, we place them ourselves on the top (in-ports) and bottom (out-ports) node boundaries. First, we place all tunnels in the middle of the node. Then, we place the remaining in- and out-ports around them. This ensures both a valid placement of the ports and some symmetry. Finally, the first and last point in the list of edge points returned by `Dagre` are replaced by points that belong to the corresponding ports.

5.2 Sugiyama-Based Algorithm

Replacing `Dagre` with our own adaptation of the Sugiyama framework enables us to make adjustments at specific steps of the framework that promote either layout quality or performance.

[1]On a side note, `Dagre` claims to be able to lay out compound graphs as described by Sander [62]; this would allow us to get a layout for all child graphs in one go. We find, however, that their implementation is broken (as of this thesis).

Recall from Section 2.3.3 that the three non-trivial steps in the framework are:

- Assigning Ranks (Section 5.3),
- Permuting Nodes Within Ranks (Section 5.4 & 5.5), and
- Assigning Coordinates (Section 5.7).

In the NGP-DAG layout problem, we additionally consider the order of ports within nodes (Section 5.6). Section 5.8 summarizes the complexity of our algorithm and the last three sections (5.9 - 5.11) describe further enhancements to our Sugiyama approach.

5.3 Rank Assignment

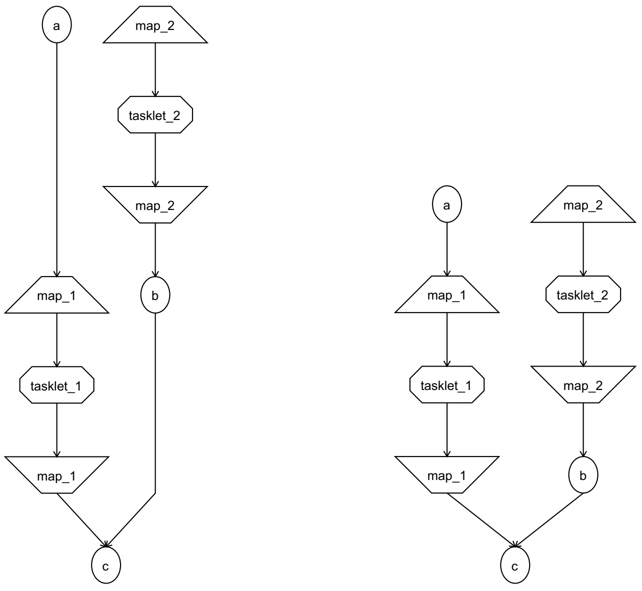
In the naive implementation of Section 5.1, the whole layout process – including the ranking – is hierarchical. That means we may have a graph with a single node sharing its rank with a complete child graph with hundreds of ranks, leading to wasted screen space (see Figure 5.1a for an example.). We propose to use global ranks [62] instead (Figure 5.1b) which can result in more compact graphs. The way to do this is to assign a child graph not just one rank, but a span of ranks, matching the number of ranks of the child graph. In order to know the number of ranks needed for a child graph, we have to rank it first and thus we have again a recursive process.

For our proposed ranking algorithm, we define the following terms: $span(G)$ is the number of ranks assigned to a graph G. If a graph G has more than one weakly connected component, the ranking is calculated separately for each of the components and $span(G)$ is set to the maximum span among the components. Similarly, the *span* of a node v is the number of ranks taken by the node, that is, $span(v) := \max\{1, span(G_{child}(v))\}$. We denote with $rank(v)$ the rank assigned to a node v, with $span(e)$ the span of edge e, that is $span((u,v)) := rank(v) - rank(u)$; and with $span_{min}(e)$ a minimum span requirement for e. To reserve enough ranks for a child graph in its parent, we set for each v the minimum span of its out-edges to $span(v)$. The *slack* of an edge is the difference of its span and its minimum span: $slack(e) = span(e) - span_{min}(e)$. An edge is *tight* if its slack is 0. A *valid ranking* of $G = (V, E)$ is one where $\forall e \in E : slack(e) \geq 0$. A *tight tree ranking* is a valid ranking such that the underlying undirected graph has a spanning tree with only tight edges. More formally, a ranking such that

$$\forall v \in V : \exists u \in N^-(v) : slack(u,v) = 0) \vee (\exists w \in N^+(v) : slack(v,w) = 0) .$$

Such a ranking is good in the sense that it has no unnecessarily long edges.

Gansner et al. [29] give an algorithm for calculating a tight tree ranking that grows a spanning tree by incrementally adding incident non-tree edges with



(a) Hierarchical Ranks **(b)** Global Ranks

Figure 5.1: Difference between hierarchical ranks and global ranks. On the left, the outer graph has 3 ranks and the inner graphs – the maps – have 3 ranks each. On the right, there is no distinction between outer ranks and inner ranks, resulting in a more compact layout.

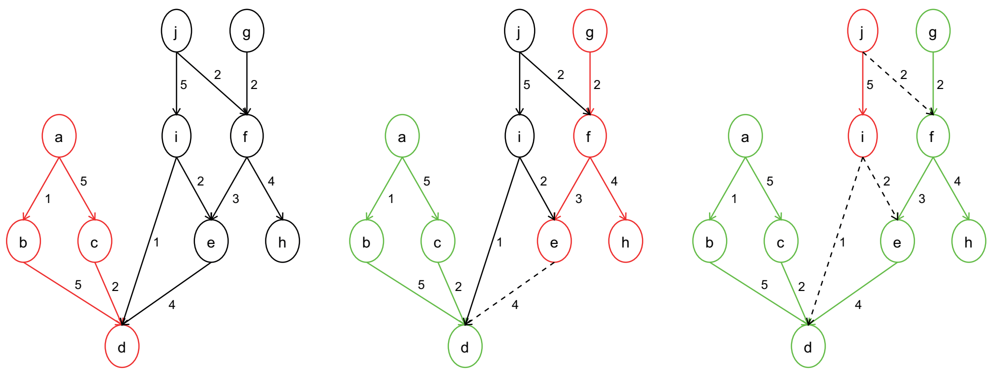
minimal slack. This implies a sorting of the edges and hence the algorithm needs $\Omega(|E| \log |E|)$ time. We present with Algorithm 2 a linear time algorithm for obtaining a tight tree ranking. We have omitted initialization for brevity but assume it is evident from context. The procedure $\textsc{Toposort}(C)$ used in line 18 returns the nodes of C in a topological order. A visualization of the algorithm can be found in Figure 5.2. We emphasize that it only works under the assumption that the input graph is weakly connected. Once it is finished, we want the overall smallest rank to be zero. To this end, we adjust the rank of each node by subtracting the currently smallest rank.

Lemma 5.1 *Algorithm 2 produces a tight tree ranking.*

Proof First, the temporary ranking within component C has a tight spanning tree by construction: Through the topological sort in line 18, a node is placed only after its in-neighbors are already placed. That is, the rank is chosen as the minimum rank that does not violate any of the minimum span constraints of the in-edges. Then, in line 22 to 26, the whole component C is placed relative to the previously ranked components of the graph. Again, we choose the minimum admissible rank considering the minimum span constraints, now given by the border edges. Eventually, all nodes are reached because every node is reachable from at least one source; and all sources are reached because we traverse all edges either upwards (lines 7 to 16) or downwards (lines 27 - 39). □

Algorithm 2 Tight Tree Ranking

```
1:  $source \leftarrow$ an arbitrary source node
2:  for $phase \leftarrow 1$ to $|sources|$ do
3:      $border = \emptyset$                        ▷ find component $C = (V_C, E_C)$ via BFS
4:      $visited[source] \leftarrow$ TRUE
5:      $q.push(source)$
6:      $C \leftarrow (\{source\}, \emptyset)$
7:      while not $q.empty()$ do
8:          $v \leftarrow q.pop()$
9:          for $w \in N^+(v)$ do
10:             $E_C \leftarrow E_C \cup \{(v,w)\}$
11:             if not $visited[w]$ then
12:                 $V_C \leftarrow V_C \cup \{w\}$
13:                 $visited[w] \leftarrow$ TRUE
14:                 $q.push(w)$
15:             if $rank[w] \neq$ NULL then
16:                 $border \leftarrow border \cup \{(v,w)\}$
17:     $rank[source] \leftarrow 0$                  ▷ temporary ranking via toposort
18:     for $v \leftarrow$ Toposort$(C)$ do
19:         for all $w \in N_C^+(v)$ do
20:             $rank[w] = \max\{rank[w], rank[v] + span_{min}((v,w))\}$
21:     if $phase > 0$ then
22:         $\Delta \leftarrow \infty$                  ▷ merge component with previous
23:         for all $(v,w) \in border$ do
24:             $\Delta \leftarrow \min\{\Delta, rank[w] - rank[v] - span_{min}((v,w))\}$
25:         for all $v \in V_C$ do
26:             $rank[v] = rank[v] + \Delta$
27:     for all $v \in V_C$ do                       ▷ add unused incident edges
28:         for all $u \in N_C^-(v)$ do
29:             if not $visited[u]$ then
30:                 $q2.push((u,v))$
31:     while not $q2.empty()$ do                   ▷ find next source via upward BFS
32:         $(v,w) \leftarrow q2.pop()$
33:         if $N^-(v) = \emptyset$ then
34:             $source \leftarrow v$
35:             break
36:         for all $u \in N^-(v)$ do
37:             if $rank[u] =$ NULL and not $visited2[u]$ then
38:                 $visited2[u] \leftarrow$ TRUE
39:                 $q2.push((u,v))$
```



(a) Phase 1
Temporary ranks:
a: 0, b: 1, c: 5, d: 7.
Final ranks:
a: 0, b: 1, c: 5, d: 7

(b) Phase 2
Temporary ranks:
g: 0, f: 2, e: 5, h: 6.
Final ranks:
g: -2, f: 0, e: 3, h: 4

(c) Phase 3
Temporary ranks:
j: 0, i: 5.
Final ranks:
j: -4, i: 1

Figure 5.2: Phases of Algorithm 2 (tight tree ranking). Node labels are chosen to reflect the order in which nodes are visited. The numbers next to the edges give their minimum span. Each phase starts at a different source of the graph (a, g, j). Red nodes and edges mark the subgraph that is ranked in this phase; green nodes have already been ranked in a previous phase. Starting at the source, all reachable nodes that are not green are added to the red subgraph. Within the red subgraph, a temporary tight tree ranking is calculated such that the source has rank 0. The red subgraph then joins the green subgraph. This is realized by incrementing or decrementing the rank of the red subgraph as a whole such that the minimum span requirements of the border edges (dashed) are met and at least one of the border edges is tight (its span is equal to its minimum span). At the end of the phase, the red subgraph and the border edges turn green and the source for starting the next phase (if there is one) is determined. This is done by walking up an uncolored edge incident to the green subgraph and from there upwards until a source is found.

Lemma 5.2 *Algorithm 2 finishes in $\mathcal{O}(|E|)$ steps* [2].

Proof Every edge can be allocated to the unique component C belonging to the phase in which the edge was first visited on a downwards path. This includes not only edges in E_C but also in *border*. Within the component, edges are visited only a constant number of times, namely in line 18, 19, 23 and 28. Before they are assigned a component, they are traversed at most twice, once in the downward BFS and once in the upward BFS; the corresponding *visited* and *visited*2 arrays are global and not local to the phase. □

5.4 Ordering Nodes

After adding dummy nodes such that every edge is between neighboring ranks, we want to reorder the nodes within each rank to reduce the number of edge crossings.

[2]Note that $|V| \in \mathcal{O}(|E|)$ because we assume the input graph to be connected.

Despite having global ranks, we do not have to order those ranks on a global scale. In fact, we want nodes that belong to the same child graph not be interleaved with nodes from other child graphs. To this end, when a child graph spans multiple ranks, we model its parent node in the parent graph by a chain of nodes. We call this chain an *intranode path* and the edges thereof *intranode segments*. We can then order the nodes of different child graphs independently from each other.

We follow the traditional [68, 29] approach of applying a one-sided crossing minimization rank by rank. That is, in the first round we assume the nodes in rank 0 fixed and reorder nodes in rank 1; in the second round we fix rank 1 and order rank 2 and so on. When we arrive at the bottom, we change direction from bottom to top. For best results, those sweeps are repeated until no further improvement can be found. We are not aware of any tight upper bound for the number of sweeps, but if X is the number of crossings in the initial ordering, then $\mathcal{O}(X)$ is a trivial upper bound. If this is a concern, one can stop the execution as soon as a sweep removes less than a certain fraction of the remaining crossings. This yields an upper bound logarithmic in the number of initial crossings. For example, if we set the threshold to 10%, after k successful sweeps the number of remaining crossings is at most $0.9^k X$ and thus after $\frac{\log X}{\log(1/0.9)} \in \mathcal{O}(\log X)$ sweeps there is at most one crossing left. Alternatively, one can also stop after a fixed number of sweeps.

It has been shown that the *barycenter heuristic* is an effective way to reduce the number of crossings in one-sided crossing minimization, often being very close to the theoretical minimum [43]. Hence our decision to employ the barycenter heuristic. The barycenter b of a node v is defined as the weighted average of the positions of the neighbors. Hence we have $b(v) = (\sum_{u \in N^-(v)} pos(u) \cdot w((u,v)))/|N^-(v)|$ for downward sweeps and $b(u) = (\sum_{v \in N^+(u)} pos(v) \cdot w((u,v)))/|N^+(v)|$ for upward sweeps where $pos(v)$ is the current position (or index) of node v within its rank and $w(e)$ is some weight attributed to edge e. In the simplest case, all edge weights are uniform (e.g., $w(e) = 1$ for all $e \in E$). Giving some edges higher weights than others increases the probability that these edges are laid out aesthetically. First, the barycenter heuristic is more likely to assign two nodes connected by a heavy edge a similar position. Second, when counting crossings (to be explained shortly), the weights of each two edges that are crossed are multiplied – making it less likely that a heavy edge is crossed. Recall that we can have more than one edge between a pair of nodes. We simplify this by removing all but one of those edges and giving it as weight the original number of edges.

It may happen that a node has no neighbors in one direction, in that case we set $b(v) = pos(v)$. The new positions are then obtained by sorting the barycenters and assigning each node the index that it has in the sorted sequence. We find that using a stable sort results in significantly fewer crossings than an unstable sort.

Changes are only applied if they provide an improvement, that is the number of edge crossings with the new order is smaller than with the previous one. More precisely, we count the number of crossings between the fixed rank and the one we want to change under the assumption of the new order. We then compare this to the previous number of crossings and adapt the new order only if it is smaller than the previous one. For counting the number of crossings between two ranks, we use the algorithm by Barth et al. [4] that has a complexity of $\mathcal{O}(|E'_r| \log(\min\{|V'_r|, |V'_{r+1}|\}))$ runtime where V'_r is the set of nodes in rank r and E'_r the set of edges between rank r and $r+1$. Recall that V' and E' are the set of nodes and edges, respectively, after adding the dummy nodes. Overall, the complexity of a downward or upward sweep is dominated by this counting complexity and thus is $\mathcal{O}(|E'| \log |V'|)$.

5.5 Resolving Conflicts

While all crossings are undesirable, we actively look to prevent two types of crossings: First, our formulation of the NGP-DAG layout problem (Section 3.3) forbids edges overlapping nodes. In principle, such overlaps are not possible in the Sugiyama framework. However, because we model nodes with child graphs as a chain of nodes (which we call intranode path) in the parent, we must ensure that no intranode path is crossed. Second, in order to have vertical inner segments (the inner part of edges that span multiple ranks, see Section 2.3.3), we prohibit inner segments crossing each other.

We call both intranode segments and inner segments of long edges *heavy segments* and all others *light segments*. A conflict arises whenever a heavy segment crosses another heavy segment (*heavy-heavy conflict*) or a light segment crosses a heavy intranode segment (*heavy-light* conflict). We call the heavy segment involved in a heavy-light conflict the *crossed segment* and the light one *crossing segment*. We denote the upper and lower nodes of a segment with a subscript n (for *north*) and s (for *south*), respectively. For example, $crossing_n$ is the upper node of a crossing segment.

Algorithm 3 resolves conflicts from top to bottom. Between each pair of adjacent ranks $(r-1)$ and r, the heavy-heavy conflicts are removed first by ResolveAllHeavyHeavy(r) and then the heavy-light conflicts one by one. Procedure HeavyLightConflict searches for a heavy-light conflict between rank $(r-1)$ and r. If there is such a conflict, the endpoints of the crossed and crossing segments are returned. If there is no conflict, the procedure returns null. In case of multiple eligible conflicts, an arbitrary one is returned.

To resolve a conflict, we have two options: First, to reorder nodes within their assigned ranks. Second, to move nodes on the y-axis; thus assigning new ranks to some nodes and inserting new dummy nodes. The second option effectively breaks the Sugiyama Framework by combining (or interleaving) the originally distinct steps 'ranking' and 'ordering'. Moreover, it leads to larger, less compact layouts. Hence we first try to apply a resolution on the x-axis (ResolveX) and only if that fails we apply a resolution on the y-axis (ResolveY).

Algorithm 3 Conflict Resolution

```
 1: procedure ResolveConflicts
 2:     $num\_ranks \leftarrow |ranks|$
 3:     for $r \leftarrow 1$ to $num\_ranks$ do
 4:         ResolveAllHeavyHeavy($r$)
 5:         while HeavyLightConflict($r$) $\neq$ null do
 6:             $conflict \leftarrow$ HeavyLightConflict($r$)
 7:             if not ResolveX($conflict$) then
 8:                 ResolveY($conflict$)
 9:                 $num\_ranks \leftarrow num\_ranks + 1$
10:     for $r \leftarrow 1$ to $num\_ranks$ do                ▷ for conflicts introduced above
11:         ResolveAllHeavyHeavy($r$)
```

5.5.1 Resolving Heavy-Heavy Conflicts

Procedure ResolveAllHeavyHeavy(r) orders all heavy segments between rank $(r-1)$ and r by their upper endpoints. The positions of the lower endpoints are then altered to reflect this order. The positions of light segments remain unmodified. See Figure 5.3 for an example.

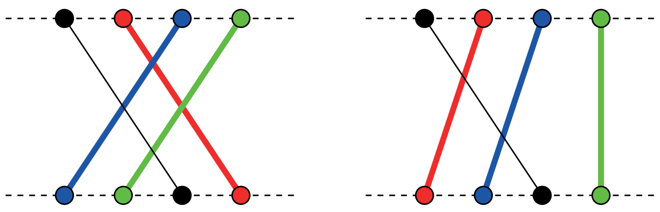


Figure 5.3: Segments between two ranks before (left) and after (right) the heavy-heavy resolution. The three colored segments are heavy and thus get reordered. The black segment is light and stays in place.

Lemma 5.3 *Assume Algorithm 3 calls the subroutine* ResolveAllHeavyHeavy *for r. Then this call removes all heavy-heavy conflicts between ranks $(r-1)$ and r and does not create any new conflicts above rank r.*

Proof After the lower endpoints of heavy segments have the same order as the upper endpoints, we can say that the heavy segments have a total order and thus, no two heavy-segments may cross. Further, segments above rank $(r-1)$ do not change their position and hence the procedure does not create any conflicts above r. □

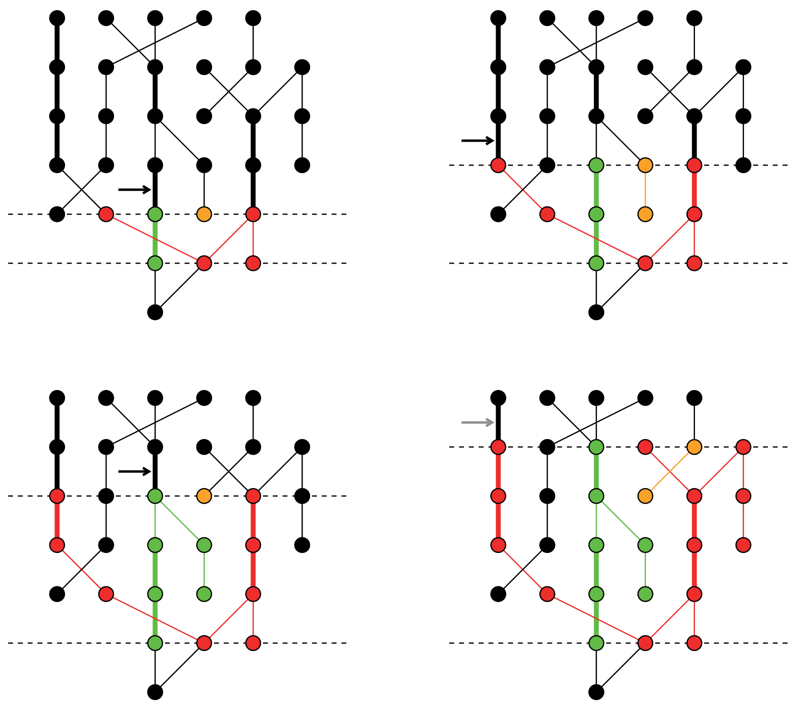
5.5.2 Resolving Heavy-Light Conflicts Horizontally

Procedure ResolveX tries to move both endpoints of the crossing segment to the same side of the crossed segment. Namely, either the right endpoint is moved to the left or the left endpoint is moved to the right. Additionally, to not immediately cause new conflicts, we move not just the endpoint, but also some of its neighbors (see below). We outline Procedure CheckResXLeft that checks whether a conflict can be resolved by moving nodes from right to left in Algorithm 4 and illustrate it with an example in Figure 5.4. It takes as input the four endpoints of the conflicting segments as found by the HeavyLightConflict procedure and returns the two sets *Green* and *Red* on success or null on failure. The procedure uses three colors to mark nodes within a certain horizontal band ('search region'). Initially this band includes the north and south rank of the conflict and is subsequently expanded upwards. Nodes connected to the crossed segment are marked green, those connected to the crossing segment red. The goal is then to move all red nodes that are right of the green nodes to their left. If a green and a red node become neighbors, we cannot safely move the red nodes and report failure. A third color, orange, is used for all nodes that have a green node to the left and a red node to the right. Their state can be seen as 'undecided': Either they will stand still with the green nodes or move together with the red nodes. Once an orange component connects to a green (red) component, its nodes change to green (red); it is no longer undecided. When the coloring is complete, orange nodes are treated as green. The search region is expanded until there is no heavy segment incident to a node at the top of the current search region, except if that node is on the left of the left-most green node.

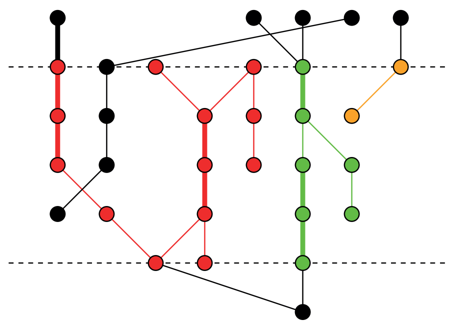
The counterpart CheckResolveXRight that tries to move nodes from left to right can easily be derived by exchanging 'left' $\longleftrightarrow$ 'right', '<' $\longleftrightarrow$ '>' and 'min' $\longleftrightarrow$ 'max where appropriate. If the resolution is possible in both directions, we choose to move the smaller set of nodes. If it is not possible in either direction, ResolveX returns false and we invoke ResolveY on the conflict.

Lemma 5.4 *Assume Algorithm 3 calls the subroutine* ResolveX *for some conflict between rank $(r-1)$ and r and* ResolveX *does not return* null. *Then this call removes the conflict and does not create any new conflicts above rank r.*

Proof Without loss of generality, assume that ResolveX moved nodes from right to left. Clearly, the conflict between segments $(crossed_n, crossed_s)$ and $(crossing_n, crossing_s)$ is gone because the red $crossing_n$ is now left of the green



(a) Expansion of the search region step by step. A black arrow indicates the heavy segment triggering the next step. In the last step, the heavy segment pointed to by the gray arrow does not expand the search region because its incident red node is on the left of any green node.



(b) Graph after moving all red nodes to the left of the green nodes.

Figure 5.4: Example of a successful x-axis conflict resolution (trying to move nodes from right to left). The horizontal, dotted lines mark the search region. Nodes adjacent to the conflicting heavy segment are marked green; those adjacent the other conflicting segment are marked red. Any uncolored node between the left-most green node and the right-most red node is marked orange. Colors are broadcast to all reachable nodes within the search region. In case of a red-orange (green-orange) conflict, red (green) takes over. If there is a red-green conflict, we stop and report that the attempted resolution failed. The search region is expanded by one level upwards if there is a heavy segment incident to a colored node at the top of the current search region that is not left to the left-most green node.

Algorithm 4 Procedure CheckResXLeft

```
procedure CheckResXLeft($crossed_n, crossed_s, crossing_n, crossing_s$)
    for all $v \in V$ do
        $root[v] \leftarrow v$
        $nodes[v] \leftarrow \{v\}$
        $color[v] \leftarrow$ NULL
    $color[crossed_n] \leftarrow$ 🟢; $color[crossed_s] \leftarrow$ 🟢
    $color[crossing_n] \leftarrow$ 🔴; $color[crossing_s] \leftarrow$ 🔴
    $expand \leftarrow$ TRUE
    $r \leftarrow rank[crossed_s]$
    while $expand$ do
        $expand \leftarrow$ FALSE
        $min_g \leftarrow \infty$
        $max_r \leftarrow -\infty$
        for all $v$ s.t. $rank[v] = r$ do
            for all $u$ s.t. $(u, v) \in E\}$ do
                if $root[u] \neq root[v]$ then
                    if $\{color[u], color[v]\} = \{$🟢, 🔴$\}$ then
                        return NULL
                    $c \leftarrow color[u]$
                    if $c =$ NULL $\vee$ ($c =$ 🟠 $\wedge\ color[v] \neq$ NULL) then
                        $c \leftarrow color[v]$
                    for all $v_i \in nodes[v]$ do
                        $root[v_i] \leftarrow u$
                        $nodes[u] \leftarrow nodes[u] \cup \{v_i\}$
                    for all $u_i \in nodes[u]$ do
                        $color[u_i] \leftarrow c$
                    if $color[u] =$ 🟢 then
                        $min_g \leftarrow \min(min_g, pos[u])$
                    if $color[u] =$ 🔴 then
                        $max_r \leftarrow \max(max_r, pos[u])$
        $r \leftarrow r - 1$
        for all $v$ s.t. $rank[v] = r \wedge pos[v] \geq min_g \wedge pos[v] \leq max_r$ do
            if $color[v] =$ NULL then
                for all $v_i \in nodes[v]$ do
                    $color[v_i] \leftarrow$ 🟠
            if $\{u \in V \mid (u, v) \in E \wedge weight[(u, v)] = \infty\} \neq \emptyset$ then
                $expand \leftarrow$ TRUE
    return $(\{v \in V \mid color[v] =$ 🟢$\}, \{v \in V \mid color[v] =$ 🔴$\})$
```

$crossed_n$ and $crossing_s$ left of $crossed_s$. Additionally, no new conflicts above the search region or within the search region have been created: Above the search region, the only segments that change their relative position are those incident to one of the top nodes between the left-most green node and the right-most red node (before moving). All those segments are light, because if one of them was heavy, the search region would have been further expanded. Therefore only light-light crossings may have been added which constitute no conflict. Within the search region, only colored segments change their relative position. After moving, all red nodes are strictly left to all green nodes and all green nodes strictly left to all orange nodes; hence no conflicts between segments of different colors. Finally, within each color, the nodes (and therefore segments) do not change their relative position, ensuring that no new conflicts arise. □

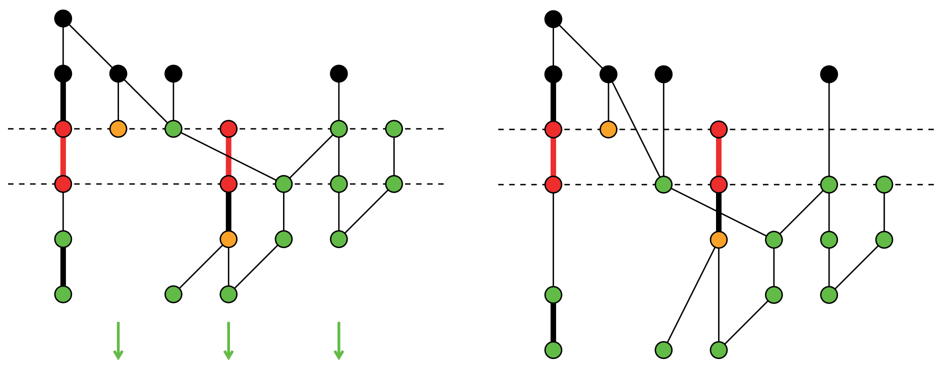
5.5.3 Resolving Heavy-Light Conflicts Vertically

Procedure ResolveY moves the crossing segment downward while the crossed segment stays in place, either moving the conflict to a higher rank or eliminating it. That is, we increase the rank of $crossing_n$ and $crossing_s$ by one. Now for the sake of preserving the hierarchy structure, we move not just these two nodes alone, but rather all nodes on or below the rank of $crossing_n$ with the following exceptions: We do not move the crossed segment, and neither any other intranode segment between rank $(r-1)$ and r. This way, we will resolve all remaining heavy-light conflicts at once. Moreover, we do not move sinks that are in rank $(r-1)$ as that would be pointless; and we do not stretch any intranode segment, that is intranode segments incident to a non-moving node stay in place. Figure 5.5 illustrates which nodes are and which are not moved.

Every edge from a non-moved node to a moved node then spans two ranks, which is a violation of the Sugiyama framework. Therefore we add for each of those edges a new dummy node in the middle rank – in place of the node that was moved – and route the edge through this new dummy node. If there is more than one in-edge for a moved node, the corresponding dummy nodes are placed next to each other in the order of the endpoints of the edges.

Lemma 5.5 *Assume Algorithm 3 calls the subroutine* ResolveY *for some conflict between rank $(r-1)$ and r. Then this call removes the conflict and does not create any new heavy-light conflicts above rank r.*

Proof The crossed intranode segment is not moved because it is on an intranode path that includes rank $(r-1)$, the rank of $crossed_n$. The crossing segment is moved because $crossing_s$ cannot be part of an intranode segment and has rank r. Hence the conflict is guaranteed to be resolved. At the same time, no new crossing above rank r is created by the procedure: The relative positions of segments between rank $(r-2)$ and $(r-1)$ do not change;



(a) Marking moving and non-moving nodes. **(b)** Graph after moving nodes.

Figure 5.5: A graph before and after moving nodes as part of the RESOLVEY procedure. Bold edges are intranode segments. The conflict is between the two ranks marked by dashed lines. Everything on and below the upper dashed line is considered to be moved one rank down. Intranode segments between the two lines (red) are required to stay in place and thus its endpoints are not moved. The orange nodes are also not moved – the higher one because it is a sink and the lower one because it is part of a non-moving intranode path. The green nodes are the ones moved which results in the situation seen on the right. At this point, multiple edges span two ranks instead of one. Each of them has to be intercepted with a new dummy node.

though two segments that have a common endpoint in rank $(r-1)$ before the procedure may have different endpoints afterwards (but by construction without introducing crossings). Between rank $(r-1)$ and r there are only two types of segments after the procedure: Intranode segments and newly added dummy segments. The intranode segments cannot cross each other (recall that all heavy-heavy crossings between rank $(r-1)$ and rank r have been removed before the procedure call); the new dummy segments are placed between the intranode segments without crossing them or other dummy segments. $\square$

Despite not creating new crossings, we may have the situation where an existing crossing becomes a heavy-heavy conflict because edges may change their weight when they turn into inner segments during the procedure. Thus we have on line 10 of Algorithm 3 an additional sweep through all ranks that will resolve any potentially created heavy-heavy conflict.

5.5.4 Correctness and Termination

Theorem 5.6 *Algorithm 3 terminates and removes all conflicts.*

Proof First, Lemma 5.3 guarantees that at line 5, all heavy-heavy conflicts between rank $(r-1)$ and r have been removed. Then, the combination of Lemma 5.4 and Lemma 5.5 guarantees that all heavy-light conflicts between those two ranks can and will be removed without creating any new heavy-light conflicts above rank r; hence the set of conflicts that can be returned by

HeavyLightConflict(r) shrinks in every iteration of the while-loop on line 5 and the loop eventually terminates. Thus, after the for-loop on line 3 all heavy-light conflicts are removed.
Next, we need to show that this for-loop also terminates. Towards a contradiction, assume that the loop in fact does not terminate, that is, *num_ranks* grows indefinitely. The variable is only increased after calling ResolveY, so we call ResolveY infinite times. Whenever ResolveY is called, it takes as input a crossed segment that is an intranode segment. Algorithm 3 does not create new intranode segments and thus, there would have to be some intranode segment that is used infinite times as an input to ResolveY with the endpoints having ever-increasing ranks. In contradiction, as soon as an intranode segment is passed to ResolveY as the crossed segment for the first time, it cannot get moved down anymore. Finally, due to Lemma 5.3 the while loop on line 10 has the invariant that after iteration r, there are no conflicts above rank r; and thus we are eventually left with a conflict-free graph. □

5.5.5 Complexity

Because the Conflict Resolution (namely ResolveY) may introduce new dummy nodes, we denote by V'' the set of all nodes and by E'' the set of all edges at the end of ResolveConflicts.

The complexity of resolving all heavy-heavy conflicts is $\mathcal{O}(|V''| \log |V''|)$ because we have to sort endpoints of heavy segments.

The complexity of searching heavy-light conflicts in our implementation is $\mathcal{O}(C \cdot |E''_{max}| + |E''|)$ where $|E''_{max}|$ is the maximum number of edges between two adjacent ranks and C is the number of reported heavy-light conflicts – including conflicts created during the resolution. This complexity stems from the fact that we search for a conflict after every resolved conflict and searching for a conflict takes $\mathcal{O}(|E''_{max}|)$ steps; and if there are no conflicts, the search in all ranks takes $\mathcal{O}(|E''|)$ steps. One could potentially improve this to $\mathcal{O}(|E''|)$ overall by marking all conflicts of a certain type between two ranks in one go and then resolve them one by one without checking for conflicts again. For a heavy-light conflict we first call CheckResolveLeft and CheckResolveRight. In the worst case, we end up exploring the entire graph and traverse $\mathcal{O}(|E''|)$ edges. Procedure ResolveX moves a subset of nodes which again can be a substantial part of the graph, hence its complexity is $\mathcal{O}(|V''|)$. In ResolveY we move almost every node below the conflict down by one rank, giving it also $\mathcal{O}(|V''|)$ complexity.
We conclude that the complexity of resolving all heavy-light conflicts is $\mathcal{O}(C \cdot |E''|)$ [3]. This dominates the complexity for searching the conflicts.

[3]Note that $|V''| \in \mathcal{O}(|E''|)$ because we assume the input graph is connected.

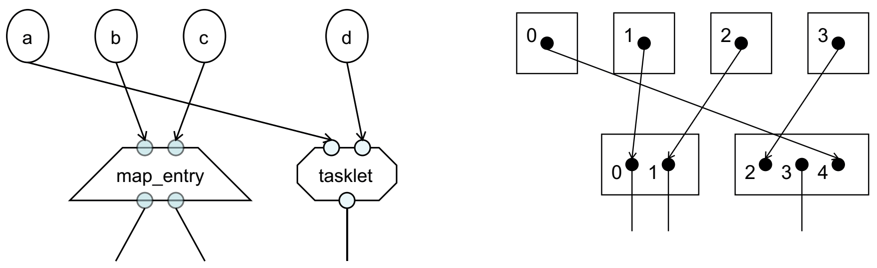


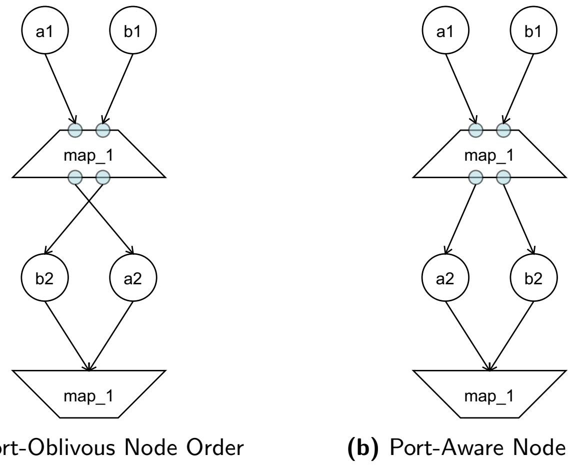
Figure 5.6: Ordering ports by the barycenter heuristic. On the left, we see two ranks of an NGP-DAG and on the right its abstraction used for ordering the ports. Observe that the ports of map_entry form tunnels; hence there are only two ports in its abstraction. The numbers next to the ports denote their position within their rank. Assume the upper rank is fixed and we reorder ports in the lower rank. The barycenters are then calculated (in order) as 1, 2, 3, 3, 0. Observe that we define the barycenter for ports without neighbors in the upper rank as its current position. For each node, the barycenters of its ports are then sorted, that is we have $[1,2]$ for map_entry and $[0,3,3]$ for tasklet. The positions are then updated as follows: $0 \to 0$, $1 \to 1$, $2 \to 3$, $3 \to 4$, $4 \to 2$.

We conclude the discussion on the conflict resolution with a remark on the origin of conflicts: Apart from crossing inner segments that give rise to heavy-heavy conflicts – that can be resolved simply and quickly – the conflicts are due to intranode segments. In turn, intranode segments originate from our goal to minimize vertical space by using a global ranking scheme. And thus, if we were to use a hierarchical ranking instead, the complexity of the conflict resolution step would be largely reduced.

5.6 Ordering Ports

So far we have not considered the order of ports, the order which is crucial to the number of edge crossings. A straightforward way to approach the issue is to first order the nodes as described above and then order the ports within their surrounding nodes. The latter works similar to the node ordering above, but now we enumerate the positions of the ports and not the nodes. This enumeration is total, see Figure 5.6 for an example. Regardless of whether a port is an in-port or out-port, we assign them the same rank. However, for tunnels we use only one of the two ports in the ordering algorithm. These ports can have both in- and out-edges the others only have edges in one direction. We then calculate the barycenter almost identical as before, for example in the downward sweep as $b(p) = (\sum_{r \in N^-(p)} pos(r))/|N^-(p)|$ where $N^-(p)$ is the set of in-neighbor ports. The ports are then reordered, but only within the range of positions assigned to the node.

Ordering the nodes first in a port-oblivious way and then ports within nodes may lead to otherwise preventable edge crossings. See Figure 5.7 for an example. It is better to order nodes and ports jointly. To this end,



(a) Port-Oblivous Node Order **(b)** Port-Aware Node Order

Figure 5.7: Comparison of a port-oblivious and port-aware ordering of the nodes. In the port-oblivious case there is no reason to put $a2$ before $b2$ and thus the original order with $b2$ on the left is preserved, leading to the crossing. When being aware of the connection to the outside nodes $a1$ and $b1$ through the tunnels, the crossing can be prevented.

we calculate the barycenter of a port p as described above with $b(p) = (\sum_{r \in N^-(p)} pos(r))/|N^-(p)|$; and the barycenter of a node v as the mean of its port barycenters: $b(v) = (\sum_{p \in p(v)} b(p))/|p(v)|$. We sort both the nodes and ports by the new barycenters. Then we first check if the new node order results in fewer crossing and if so, store the new node order. Then, we check for each of the nodes, if reordering its ports according to the new barycenter sequence marks an improvement and if so, permute the positions of ports within the nodes. This is essentially the method described by Schulze et al. [65]. We still have to do the conflict resolution described in the previous section. We even find that the final result is better if we order the nodes again in the port-oblivious way described in Section 5.4 before resolving conflicts. The reason might be that the number of conflicts is reduced by this – observe that conflicts arise from the order of *nodes* but not the order of *ports*. We hence see the joint ordering of nodes and ports just as an optional preprocessing step to the ordering process. We outline the complete ordering process as procedure DoOrder in Algorithm 5. The argument *order* is an abstraction of the order of both nodes and ports including the rank of each node. It is mutated by each of the subroutines JointlyOrderNodesPorts, OrderNodes, ResolveConflicts and OrderPorts. Procedure OrderNodes takes as argument a boolean value that indicates whether attempts to create a conflict should be stopped. This is necessary when reordering nodes again in line 5 after resolving the conflicts in line 4. The conflicts can be prevented by adapting a new permutation only if there are no conflicts induced by it.

Algorithm 5 Ordering Nodes and Ports

```
1: procedure DoOrder(order)
2:     JointlyOrderNodesPorts()                         ▷ optional
3:     OrderNodes(false)
4:     ResolveConflicts()
5:     OrderNodes(true)          ▷ prevent conflicts during ordering
6:     OrderPorts()
7:     return order
```

5.7 Assigning Coordinates

Assigning the nodes' y-coordinates is trivial. We have a list of the nodes on each rank and set the ranks' height to the maximum height of a node. If a node has a child graph, its height has to be calculated first by assigning y-coordinates. Then we set the distance between two ranks as the ideal edge length d.

For assigning the x-coordinates we adapt an algorithm by Brandes and Köpf [11]. Their algorithm tries to vertically align each node with a neighbor; more specifically, with the middle (or median) neighbor – or one of two middle ones if the number of neighbors is even. Because dummy nodes only have one neighbor on each side, inner segments are going to be laid out vertically – if they can be aligned with that neighbor. Note that we are not allowed to change the order of nodes; and thus if there is another edge that crosses the inner segment and that other edge has to be vertical, the inner segment cannot be vertical. To mitigate this, the algorithm does not allow inner segments to be crossed at all. Algorithm 6 marks all edges that cross heavy edges – edges with infinite weight, that is inner segments and intranode segments – as unusable. Only light edges can be marked because heavy-heavy crossings are already resolved before as described in Section 5.5. The variable *ranks* is a list of lists such that $ranks[r]$ is the list of all nodes in rank r ordered by their position calculated in the previous step of the framework. We denote with v_r^n the n-th node in rank r, that is $ranks[r][n]$.

The algorithm of Brandes and Köpf then tries to align each node with one of its median neighbors. They do this in four different ways that they call 'leftmost upper', 'rightmost upper', 'leftmost lower' and 'rightmost lower'. *Upper* refers to aligning nodes with upward neighbors and *lower* to aligning them with downward neighbors. *Leftmost* and *rightmost* denote the vertical direction in which conflicts are resolved. This yields four different layouts that are then merged into the final layout. For each node, each of the four layouts defines an x-coordinate, so there are four suggestions for where to place a node horizontally. The x-coordinate in the final layout is then calculated as the median value of these four suggestions.

Algorithm 6 Marking Conflicts [11]

```
1: for all $e \in E'$ do
2:     $usable[e] \leftarrow$ TRUE
3: for $r \leftarrow 1$ to $|ranks| - 1$ do
4:     $left \leftarrow -1$
5:     $n \leftarrow 0$
6:     for $k \leftarrow 0$ to $|ranks[r]| - 1$ do
7:         $heavy \leftarrow \exists u \in N^-(v_r^k) : w((u, v_r^k)) = \infty$
8:         if $k = |ranks[r]| - 1$ or $heavy$ then
9:             $right \leftarrow |ranks[r-1]|)$
10:            if $heavy$ then
11:                $right \leftarrow \min_{u \in N^-(v_r^k)} pos[u]$
12:            while $n \leq k$ do
13:                for all $u \in N^-(v_r^n)$ do
14:                    $usable[(u, v_r^n)] \leftarrow pos[u] \geq left$ and $pos[u] \leq right$
15:                $n \leftarrow n + 1$
16:            $left \leftarrow right$
```

We next describe how to generate those four layouts on the example of the leftmost upper layout. The other three can easily be derived by symmetry.

For generating the leftmost upper layout, we go from top to bottom and try to match each node with one of its median upward neighbors. The reason we say *try* is twofold: First, each node in the upper rank can only be aligned with at most one node in the current rank. But multiple nodes can have the same upper node as median neighbor. And second, the edges of the matching must be chosen such that no two edges cross – because the crossing edges cannot be both drawn vertically without changing the node order. To this end, the algorithm goes from left to right and assigns each node one of its upward median neighbors only if the position of that neighbor is larger than any of the positions of previously assigned neighbors. If a node has two median neighbors, we first try the left and then the right one.
This process groups the nodes into *blocks* such that the centers of nodes in the same block are horizontally aligned. A block is a chain of nodes on consecutive ranks but can also be just a single node. Then, an x-coordinate is assigned to each block such that the blocks are placed in a non-overlapping but compact way. This is essentially done by arranging the blocks in another graph, called *block graph* and then calling a ranking algorithm on it. In the block graph, each node represents a block and horizontally neighboring blocks are connected by an edge in the block graph. The minimum span of each edge (b_1, b_2) is set such that the space between b_1 and b_2 is equal to some distance s. That is, $span_{min}((b_1, b_2)) := (width(b_1) + width(b_2))/2 + s$ where

width(b) is the maximum width of a node in block b. The rank assigned to a block b by the ranking algorithm then gives the x-coordinate of the center of b. Algorithm 7 shows how to find blocks and arrange them in the block graph. For the ranking algorithm, Brandes and Köpf [11] adapt the longest path algorithm (see Section 2.3.3). We propose to use our tight tree ranking algorithm from Section 5.3 instead, that produces more compact rankings than the naive longest path algorithm.

Algorithm 7 Building the Block Graph [11]

```
1: $B = (\emptyset, \emptyset)$                                   ▷ block graph $B = (V_B, E_B)$
2: for $r \leftarrow 1$ to $|ranks| - 1$ do
3:     $min \leftarrow 0$
4:     for $n \leftarrow 0$ to $|ranks[r]| - 1$ do
5:         $match \leftarrow$ NULL
6:         if $|N^-(v_r^n)| > 0$ then
7:             for $m \leftarrow \lfloor \frac{|N^-(v_r^n)|}{2} \rfloor, \lceil \frac{|N^-(v_r^n)|}{2} \rceil$ do
8:                 if $usable[neighbors[m]]$ and $pos[neighbors[m]] \geq min$ then
9:                     $neighbor \leftarrow neighbors[m]$
10:                    $min \leftarrow m + 1$
11:                    break
12:        if $neighbor =$ NULL then
13:            $V_B \leftarrow V_B \cup \{b\}$                        ▷ $b$ is a new block
14:            $block[v_r^n] \leftarrow b$
15:        else
16:            $block[v_r^n] \leftarrow block[neighbor]$
17: for $n \leftarrow 1$ to $|ranks[r]| - 1$ do                   ▷ add edges between blocks
18:     $E_B \leftarrow E_B \cup \{(block[v_r^{n-1}], block[v_r^n])\}$
```

After trivially setting the nodes' y-coordinates, and their x-coordinates through the algorithm of Brandes and Köpf, we can next place the ports, and finally the edges.

Ports are horizontally centered within their parent – or rather, as much centered as their order allows. For example, we might have two tunnels and between them one in-port and two out-ports. In that case, the distance between ports cannot be uniform.

Assigning the edge coordinates is straightforward once the node and connector coordinates are settled, as an edge starts at the bottom center of its origin port, potentially goes through dummy nodes (points) and ends at the top center of its destination port. However, there is one detail that calls for attention: If nodes of a rank have non-uniform height, the straight line between the origin and destination may overlap some node – which is something we

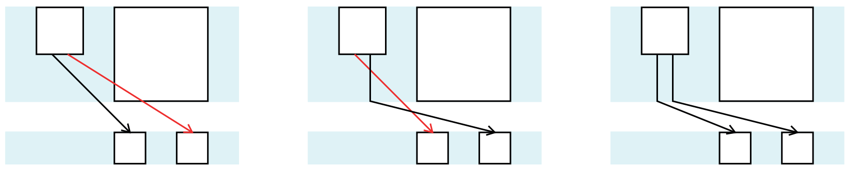


Figure 5.8: Preventing edge overlaps through proxy points. The shaded regions mark the ranks. We align nodes of the same rank at the top. In (a), edges go straight from origin port to destination port, resulting in a node overlap. In (b), the problematic edge is routed through a proxy point at the rank bottom which resolves the node overlap. However, there are now two edge crossings. By adding a proxy point for all out-edges as in (c), we do not create this kind of crossings.

prohibit. See Figure 5.8 for an illustration of the problem. To prevent it, we may ensure that edges run vertically between the rank boundaries. This can be done by adding *proxy points* with the same x-coordinate as the origin or destination port at the top or bottom rank boundary. We do not want to do this for all edges by default because proxy points constitute bend points which are undesirable. Therefore we check for each edge if it overlaps a node and only if that is the case, we add the corresponding proxy point. Yet, when some out-edges of a node are proxied and some are not, this may induce additional edge crossings (see Figure 5.8). We hence add proxy points for all the out-edges of a node, or none.

5.8 Complexity Overview

The only part of our Sugiyama implementation whose time complexity is not in $\mathcal{O}(|V''| + |E''|)$ is the ordering phase, including the conflict resolution. On one hand, we have $\mathcal{O}(S \cdot (|E''| \log |V''|))$ for the ordering of ports and nodes where S is the number of up- and downward sweeps. On the other hand, we have $\mathcal{O}(|V''| \log |V''| + C \cdot |E''|)$ for resolving all conflicts where C is the number of heavy-light conflicts. As we shall show in Chapter 8, in practice S and C are small enough for the layouter to finish in reasonable time, even for large graphs.

There is an additional observation that helps to alleviate the overall complexity of our algorithm: We do not need to order all nodes of the NGP-DAG at once, but only those that form a connected component. In particular, isolated child graphs can be ordered independently.

5.9 Shuffling the Input

When ordering nodes (Section 5.4) and ports (Section 5.6) with the barycenter heuristic, we iteratively apply changes that reduce the number of crossings but discard them if they do not reduce it. Hence, the method is prone to converge to local rather than global minima. One way to counter that is to start the ordering process from multiple, random permutations of the ranks and then choose the result with the smallest number of crossings. This was also proposed by Jünger and Mutzel [43]. Because we do not use the node and port order in a previous step, this is equivalent with shuffling nodes and ports in the input. We emphasize however that the order given in the input may reflect a natural order that is already close to the optimum. We hence suggest to always use the input order as one of the considered initial orders.

Algorithm 8 shows the shuffling process around the ordering phase. Procedure CountCrossings(*order*) sweeps through the layers and counts all crossings induced by *order* with the algorithm by Barth et al. [4]. Procedure Shuffle(*order*) returns a random permutation of *order*.

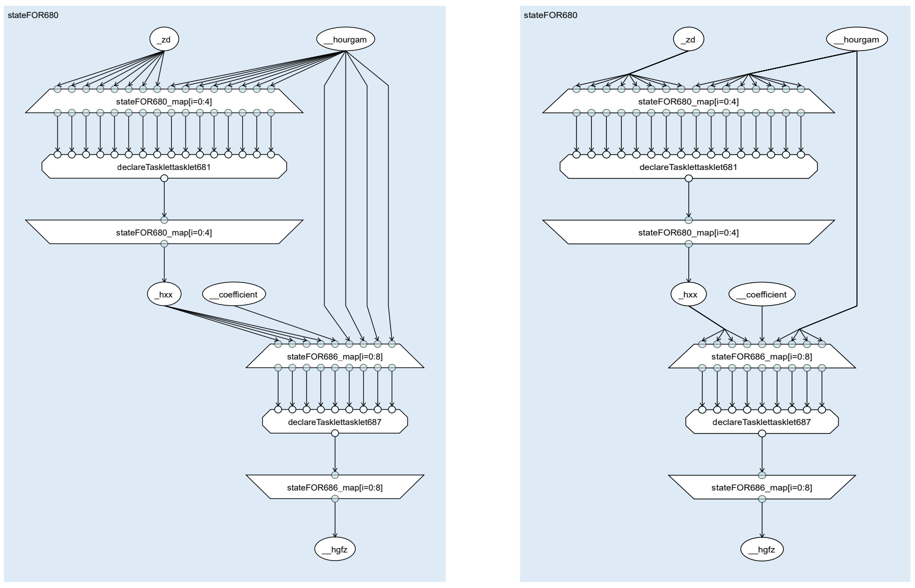
Algorithm 8 Permuting the Input

```
1: $order_{min} \leftarrow$ DoOrder($order_{init}$)          ▷ initial order is always first guess
2: $c_{min} \leftarrow$ CountCrossings($order_{min}$)
3: for $i \leftarrow 1$ to $s$ do                          ▷ try $s$ random permutations
4:     $order \leftarrow$ DoOrder(Shuffle($order_{init}$))
5:     $c \leftarrow$ CountCrossings($order$)
6:     if $c < c_{min}$ then
7:         $c_{min} \leftarrow c$
8:         $order_{min} \leftarrow order$
```

5.10 Crossing Angle Optimization

We have established in Section 2.3.1 that the angle at which two edges cross, is relevant to the quality of a layout – and that they ideally cross at a right angle. We then observe that we can change the crossing angles by stretching the involved edges vertically. More specifically, by choosing a good vertical distance between two ranks, we might improve the angle cost of all crossings between the two ranks. On the other hand, the distance between two ranks should not be too far off from the ideal edge length d due to additional cost implied by longer edges. We therefore search for a distance that minimizes the sum of the angle cost and the length cost. For aesthetic reasons we decide to restrict the allowed distances to the range $[d, 2d]$ [4]. Then we search numerically for the minimum within this interval.

[4]Our cost function is not perfect. For example, when all edges between two ranks have length d but are very 'flat', the ranks appear to be too close to each other.



(a) Without Bundling **(b)** With Bundling

Figure 5.9: Example of edges that can be bundled. The bundling increases readability by reducing the number of drawn segments.

That is, we linearly partition the interval and evaluate the cost at each point [5]. In practice, we evaluate it for every pixel between d and $2d$.

5.11 Edge Bundling

In some cases, SDFGs have more than one edge between the same two nodes. More specifically, this may happen when a data stream between a container and an execution unit is divided into multiple parts. In that case we can consolidate these parts into one edge that is split only at the very end. We call such an edge a *bundle*. See Figure 5.9 for an example. It is essential for comprehensibility that all ports of a bundle lie next to each other. This can be achieved in a postprocessing step where the position of a *bundle of ports* is determined by the average position of its ports.

The Sugiyama framework is a level-based approach to create upward drawings of DAGs. We have shown how to adapt it to the NGP-DAG model and suggested some extensions that may improve the quality of the layouts. In the next chapter, we look at a fundamentally different approach.

[5] Note that there can be up to two local minima and a binary search would not necessarily find the global one.

Chapter 6

Force-Directed Approach

Other than the Sugiyama framework, force-directed approaches do not divide the nodes into levels. Instead, nodes can move freely in the plane, based on forces the nodes exert on each other.

6.1 Magnetic Spring Model

We see the *magnetic spring model* by Sugiyama and Misue [67, 66] as a good candidate of a force-directed solution to the NGP-DAG problem. First, it is straightforward to implement and second, it can be easily adapted or extended to meet our needs. For a rough evaluation of the layouts we can obtain through this method, we ignore the existence of ports for the main part of the algorithm and add them only after the nodes are placed. This is the same as in our `Dagre`-based solution (see Section 5.1).

There are three forces in the model:

- *Magnetic force*: In the magnetic spring model [67], edges are modeled as magnetic springs. That is, they are exposed to a rotative force in the direction of some magnetic field. In our application, we use a parallel magnetic field pointing downwards, $m = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$. The direction of the force is the downward normal vector of the edge, see Figure 6.1. The strength is defined as $b|\overrightarrow{u_b v_t}|^{\alpha}\theta^{\beta}$ where b is the magnetic strength constant, θ the angle between the edge and the magnetic field and α, β some constants. We find that choosing $\alpha = 1$ makes sense because that rotates two edges with equal angle θ by the same amount, irrespective of their length. For the field strength we can also choose $b = 1$, because we will later scale the force in any case. Regarding the angle factor, we request a change to the base of exponentiation: For an edge (u, v), we want the magnetic force to pull down v if the edge is directed upwards. If (u, v) is already directed downwards, the magnetic force may be relatively small. We

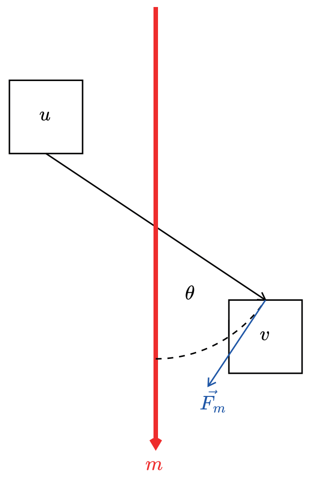


Figure 6.1: Magnetic force $\vec{F_m}$ exerted on edge (u, v) by magnetic field m. The angle θ between the edge and m decides the length of $\vec{F_m}$.

thus propose to use $(\theta/(\pi/2))^\beta$ as the strength of the magnetic force. Note that $\theta \in [0, \pi]$ and hence $\theta/(\pi/2) \in [0, 2]$. A second deviation from the original model is that we apply the magnetic force only on one of the end nodes (the destination node). This helps nodes with both in- and out-neighbors to actually move down when they are above their in-neighbors. The complete magnetic force on node v then is

$$\vec{F_m}(v) := \sum_{u \in N^-(v)} \mathrm{n}(\overrightarrow{u_b v_t})(\theta(\overrightarrow{u_b v_t})/(\pi/2))^\beta$$

where $\mathrm{n}(\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right]) = \left[\begin{smallmatrix}-\mathrm{sgn}(y)(1/x)\\ \mathrm{sgn}(y)(1/y)\end{smallmatrix}\right]$ is the downward normal vector of same length and $\theta(\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right]) := (\pi/2) - \mathrm{sgn}(y)\arctan(|y/x|)$ is the angle to the magnetic field $m = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$.

- *Spring force*: If an edge is longer than the ideal distance d, the adjacent nodes are pulled towards each other. If the edge is shorter, the nodes repel each other. We use the logarithmic definition as given in the original paper [67] [1]. We find however that applying the force unidirectionally (same as the magnetic force) produces better results. We thus have

$$\vec{F_s}(v) := \sum_{u \in N^-(v)} \overrightarrow{u_b v_t} \cdot \frac{\ln(|\overrightarrow{u_b v_t}|/d)}{|\overrightarrow{u_b v_t}|}$$

where $\overrightarrow{u_b v_t}$ is the vector from the bottom center of u to the top center of v.

[1] A direct comparison of this logarithmic force with Hooke's Law by Klapaukh et al. [45] suggests that the logarithmic might generally perform better in terms of overlaps and edge crossings.

- *Repulsive force*: Nodes are modeled as charged particles that repel each other according to Coulomb's law $F = k\frac{q_1 q_2}{r^2}$ where q_1 and q_2 are the charges, k is a constant and r the distance between them. The charge is considered to be uniform and the force is not applied to neighbors – because with the spring force they already have a repulsive or attractive force between them. We differ from the original formulation by the following observation: The square distance in the denominator makes the force becomes quickly negligible when it is larger than 1. Presumably the distance is measured in absolute terms, for example pixels. However, for small, ¿ 1 pixel distances, we still want to have a strong repulsion. We therefore suggest to define r as the distance relative to some ideal distance s. (Note that the ideal distance between non-neighbors s may differ from the ideal edge length d.) We then arrive at the following definition of the repulsive force:

  $$\vec{F}_r(v) := \sum_{u \in V \setminus N(v)} \overrightarrow{uv} \cdot \frac{1}{(|\overrightarrow{uv}|/s)^2}$$

  where $\overrightarrow{uv}$ is the distance between the nodes – the distance between the node boundaries when drawing a line from center to center [2].

The repulsive force $\vec{F}_r(v)$ can by definition become arbitrarily large. The magnetic force $\vec{F}_m(v)$ may also be undesirably strong for long edges and $\beta > 1$. We therefore put a limit c on all three forces. After that, we calculate the net force $\vec{F}(v)$ as a linear combination where each of the forces is multiplied by a weight (w_m, w_s, w_r respectively).
We use an additional heuristic proposed by Harel and Koren [33]. They found that the repulsive forces between non-neighbors – that prevents nodes from overlapping or generally being too close – may promote unpleasing local minima. This can be overcome by starting the layout process with just the other two forces and adding the repulsion in later iterations. To this end, we set $w_r = 0$ initially and then linearly increase it with each iteration.

The overall algorithm is given as Algorithm 9. Initially, we place all nodes on a circle in random order. Then, in every iteration we calculate for each node v the net force $\vec{F}(v)$ as described above and update the node's position ($pos[v]$) by adding $\vec{F}$. We stop after a fixed number of iterations (T) or when the layout has converged, that is none of the nodes moved more than $\varepsilon > 0$ in one iteration.

The worst-time complexity of Algorithm 9 is trivial. Assuming that we do not stop early because of convergence, we have $\mathcal{O}(T \cdot |V|^2)$.

[2] If the distance would be 0 or negative because the nodes are already overlapping, we set it to a very small value like 10^{-5}.

Algorithm 9 Magnetic Spring Layout

```
 1: set $pos[v]$ randomly for all $v \in V$
 2: $w'_r \leftarrow 0$
 3: for $i = 1$ to $T$ do
 4:     for all $v \in V$ do
 5:         calculate $\vec{F}_m[v]$, $\vec{F}_s[v]$, $\vec{F}_r[v]$ as described above
 6:         for $\vec{F}_* \leftarrow \vec{F}_m, \vec{F}_s, \vec{F}_r$ do                ▷ limit forces to $c$
 7:             $\vec{F}_*[v] \leftarrow \frac{c\,\vec{F}_*[v]}{\max\{|\vec{F}_*[v]|, c\}}$
 8:         $w_m\vec{F}_m[v] + \vec{F}[v] \leftarrow w_s\vec{F}_s[v] + w'_r\vec{F}_r[v]$      ▷ calculate net force $\vec{F}$
 9:         $pos[v] \leftarrow pos[v] + \vec{F}[v]$                          ▷ apply net force
10:     if $\max_{v \in V} \vec{F}[v] \leq \varepsilon$ then                       ▷ check convergence
11:         break
12:     $w'_r \leftarrow w'_r + \frac{w_r}{T}$                                 ▷ increase repulsion
```

With the magnetic spring model we have a viable alternative to the Sugiyama framework of the previous chapter. Before we compare the two approaches in Chapter 8, the following Chapter 7 is dedicated to the implementation.

Chapter 7

Implementation

We design our layout algorithms to be run as part of a complete web application, the *SDFG Viewer* (SDFV) [63]. The SDFV takes an SDFG as input, calculates a layout for it and draws it on screen. It further allows users to interact with the drawing, for example by showing additional information when moving the cursor over nodes or edges. For our implementation we stick to JavaScript (JS), the language of the SDFV. This also has the advantage that the layouter can be run with any standard web browser. More precisely, we choose to write it in Typescript (TS) and compile to JS for the ease of development [10].

As speed is a crucial requirement for the use in an interactive tool, we seek to improve the performance of our algorithms other than what can be accomplished through algorithmic changes. In the following, we describe the implementation of two ideas towards this goal. We apply these ideas only to the Sugiyama-based layouter but not to the magnetic spring layouter since we rate the quality of drawings generated with the Magnetic Spring Layouter as insufficient (see next chapter).

7.1 WebAssembly

Recently, WebAssembly (WASM) [32] has been introduced as a portable, low-level bytecode that is supported by all major browsers and claims to achieve near native performance.
We use *Emscripten* [1] to compile code from C to WASM. Porting the entire Sugiyama layouter – a few thousands of lines – would be plenty of work. Instead, we choose to rewrite just one of its most time-consuming subroutines, the permutation of nodes and ports by the barycenter heuristic. Emscripten not only generates the WASM file but also a JS bridge so that we only have to call a certain function, that takes the ranked graph as argument, in our TS code.

7.2 Multithreading

Today's consumer computers typically include processors with 4 to 24 hardware threads. Using just one of them would be suboptimal when aiming for high performance.
Conveniently, JS provides an API for multithreading, the *Web Workers API*. The question then is, which parts of the algorithm should be parallelized.

Recall that one of our improvements of the Sugiyama framework is to start the node and port ordering from different, random permutations and choose the one with the best result (see Section 5.9). Running these in parallel appears to be a natural choice, as we expect the ordering to take similar time for all permutations of the same graph – and thus, to improve the final layout without a major time penalty.

A second straightforward part to parallelize is the Brandes-Köpf algorithm that merges coordinate assignments obtained via four different, independent runs of Algorithm 7 (see Section 5.7). We simply map each of the four runs to a dedicated worker thread.

In the next chapter we will evaluate how our two algorithms perform and to what extent the use of WASM and multithreading benefit the performance.

Chapter 8

Evaluation

We evaluate the different layout algorithms with respect to quality and performance. Regarding quality, we are primarily interested in the fulfillment of the constraints we define in Section 3.3 and secondarily in the cost of the generated layouts as defined in Chapter 4.

8.1 Experimental Design

We use SDFGs from the *NPBench* suite [79] and few other graphs provided by real-world SDFG-based applications [64, 38, 77]. Our largest graphs stem from climate modeling [2]. We divide the graphs into the following test sets:

- POLY: 30 graphs from the *polybench* subset of NPBench. These graphs are relatively small with $|V|$ ranging from 7 to 353 (median 31.5).
- TALL: 2 graphs (*bert2*, *yolov4-fused*) which are highly sequential and have long paths.
- WIDE: 3 graphs (*bert*, *eos*, *linformer*) that have a high level of parallelism and are therefore prone to be drawn wide.
- PORT: 3 graphs (*deriche2*, *lulesh*, *va-gpu*) with a high port-to-node ratio.
- DSW1: 2 graphs for the same climate model [2], once highly sequential (*d_sw1*, $|V| = 39,157$ and $|E| = 35,014$) and once highly parallel (*d_sw1-fused*, $|V| = 27,584$ and $|E| = 37,468$).

All runtime measurements were conducted on a consumer-grade notebook with an Intel Core i7-8650U processor (4 cores at 1.90 GHz, Turbo Boost disabled). If not stated otherwise, the layouter was run through Google Chrome (version 92.0.4515). Reported times are median values from at least 5 runs. Between two measurements the browser was always restarted. This is a necessity to get consistent results, since JavaScript engines (and especially

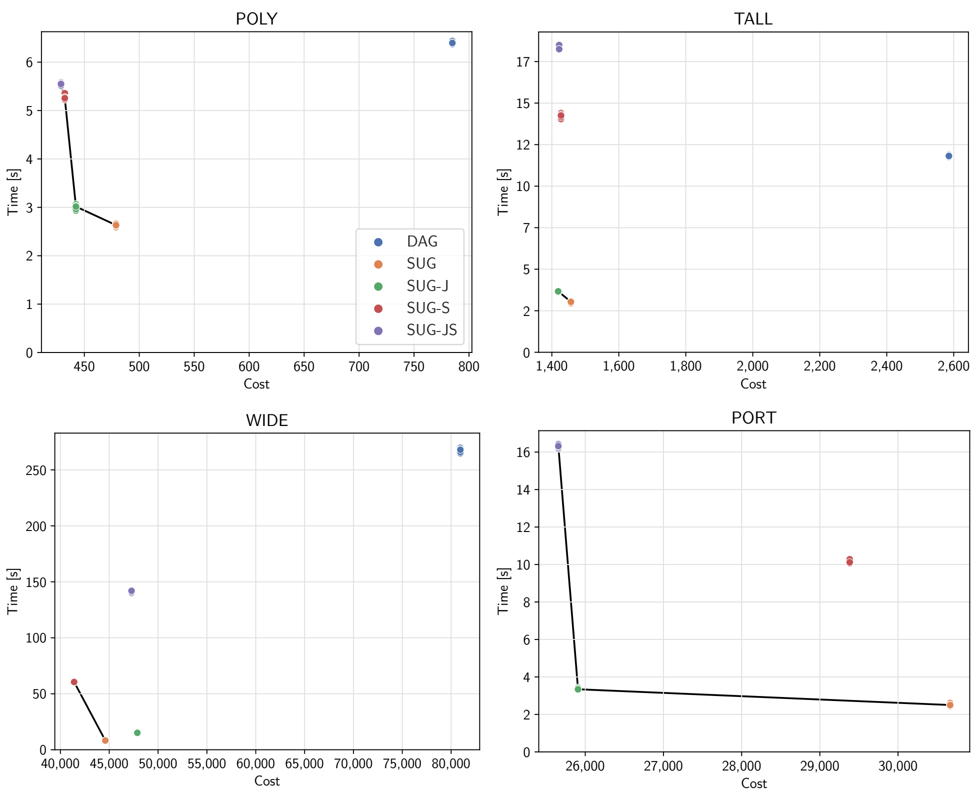


Figure 8.1: Quality and performance of our layout algorithms on different test sets. The indicated cost and time are both summed over all graphs of the respective set. The black lines mark the Pareto frontier.

Chrome's *V8* engine) optimize code during execution and subsequent calls to the layouter would otherwise be faster. We will briefly explore this effect in the discussion about WASM below.

8.2 Level-Based Approach

In Figure 8.1, we plot the cost of generated layouts against the time needed to generate them for several layouters.
DAG refers to the `Dagre`-based layouter from Section 5.1. *SUG* is our own implementation of a Sugiyama-based layouter (rest of Chapter 5). The suffix *J* stands for including the joint ordering of nodes and ports as described in 5.6. The suffix *S* means that the input has been shuffled 10 times (see 5.9).
The `Dagre` layouter may produce layouts where edges overlap nodes. This is not only a constraint violation, but also makes the cost function invalid – because it affects the number of edge crossings. We thus exclude graphs from the POLY set that would be drawn with such overlaps. This is only the case for 3 out of 30 graphs, namely *adi*, *durbin* and *syr2k*.

On these sets, our SUG implementation is 2.4 to 32.5 times faster than DAG while reducing the cost by 39 to 49 percent. The joint ordering step does not typically induce much extra time but may reduce the number of crossings and thus the cost significantly. Shuffling also seems effective – at least when omitting the joint ordering – but the runtime is roughly proportional to the number of permutations.

`Dagre`-Based vs. Sugiyama-Based Layouter As mentioned before, the `Dagre` library [55] does not prevent node-edge overlaps. An example of such an overlap can be seen in Figure 8.2a. This is a violation of our layout constraints defined in Section 3.3. Further, the cost of layouts generated with DAG tend to be high due to ports not being ordered to minimize crossings and edges having at least one bend when they are not vertical. The only advantage of the DAG over SUG is that some graphs are laid out more compact. DAG uses the network simplex algorithm [29] that minimizes the number of dummy nodes while SUG uses our tight tree ranking algorithm which gives no such guarantee.

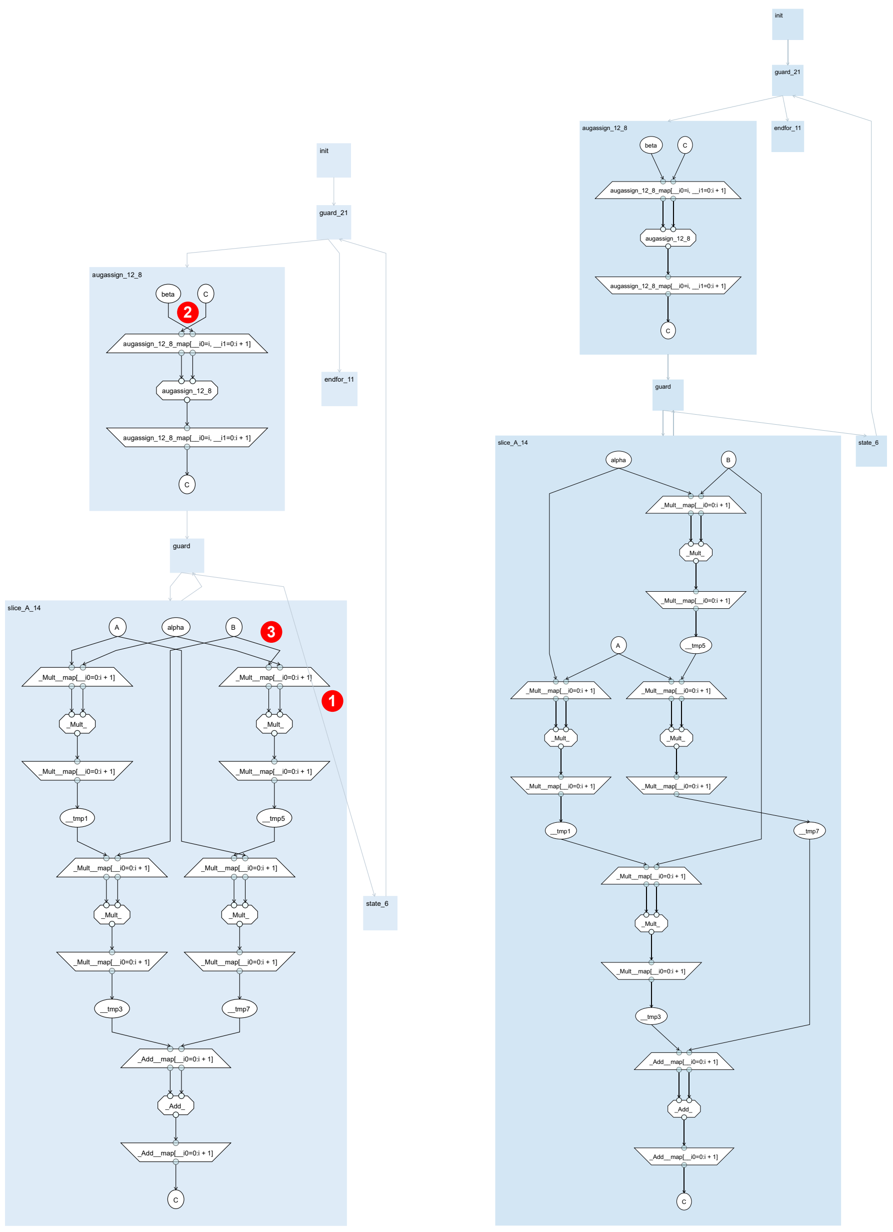
Our adaptation of the Sugiyama framework on the other hand has both a mechanism that forbids node-edge overlaps (see Section 5.7), and a heuristic for ordering ports. As a consequence, SUG not only meets our hard constraints but also produces layouts with lower cost than DAG. Moreover, as seen in Figure 8.1 SUG is significantly faster than DAG.

Next, we will highlight how certain parts of the algorithm affect the quality and the performance.

Ranking We propose to use a global ranking in spite of generating more compact layouts. Table 8.1 compares the total number for ranks using this approach with the number of ranks we would get from a hierarchical ranking. We find the largest reduction of ranks in the POLY set (13%). This corresponds to the fact that ranks can only be saved when there are child graphs running parallel. For the PORT set, the global ranking induces *more* ranks than the hierarchical ranking. This is due to a large number of y-based conflict resolutions (which add extra ranks) for one of its graphs (*deriche2*).

Table 8.1: Number of ranks using hierarchical versus global ranking (SUG-J).

Test set	Hierarchical	Global
POLY	866	756
TALL	2094	2136
WIDE	1515	1515
PORT	334	346



(a) Dagre-Based (DAG) **(b)** Sugiyama-Based (SUG-J)

Figure 8.2: Drawing of *syr2k* with Dagre and with our own implementation of the Sugiyama framework. The Dagre-based solution creates a node-edge overlap (1), unnecessary edge crossings (2) and bends (3). Both rankings have a tight spanning tree, but the one on the left – calculated with the network simplex algorithm – has fewer ranks and (by coincidence) highlights the symmetry inherent to the graph.

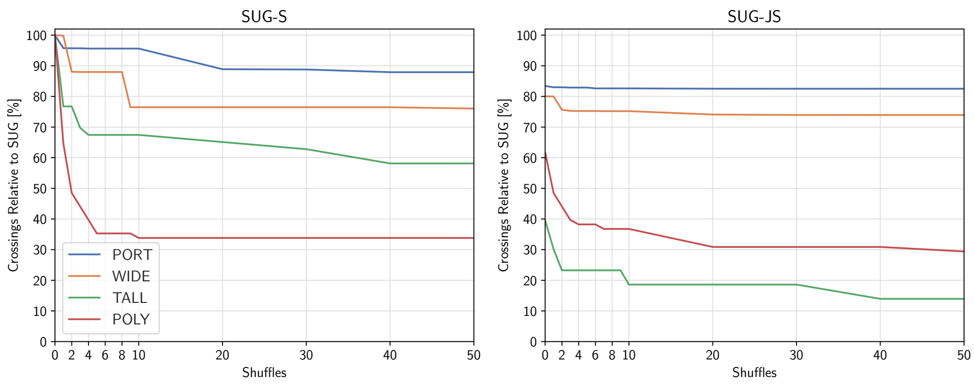


Figure 8.3: Effect of shuffling on the number of edge crossings (with and without jointly ordering nodes and ports). The crossings are summed over all graphs of the respective set.

Shuffling We next want to indicate the impact of shuffling in more detail. Recall that we randomize the order of both nodes and ports towards circumventing local minima in the barycenter heuristic. Figure 8.3 shows by how much the number of crossings may be reduced by increasing the number of shuffles. We find that the relative reduction is larger when the absolute number of crossings is smaller. With SUG-S, the number of crossings for the POLY set is cut in half with only two shuffles and to roughly one third with 10 shuffles. Except for the TALL set, the effect with SUG-JS is smaller. This is no surprise as the number of crossings with SUG-J is already closer to the theoretic minimum than with SUG.

Note that a large reduction in the number of crossings does not necessarily decrease the cost significantly. In particular, when the crossings make only for a small fraction of the total cost, the shuffling can improve the total cost only marginally. For example, observe that SUG-JS with 10 shuffles reduces the number of crossings for the TALL set by more than 50 percent, but the cost of SUG-JS is very close to SUG-J in Figure 8.1.

Performance We have given some upper bounds on the runtime complexity in Section 5.8. However, those bounds are rather imprecise and thus we additionally evaluate the time needed for each step of the algorithm in practice. The result can be seen in Figure 8.4. The total time for generating a layout depends primarily on the size of the graph, that is, $|V|$ and $|E|$. For example within the POLY set, the largest graphs are *deriche*, *adi* and *heat_3d* – for which we measure the highest time. The ranking step is always the fastest in our tests and completes in less than a second for all tested graphs.
The conflict resolution phase is also very short for most graphs, because heavy-light conflicts are rare in our tests. In the POLY set they appear in only 3 graphs with a total of 4 conflicts and for the WIDE and TALL set we encounter only 1 heavy-light conflict. Graph *deriche2* has 16 of them, which

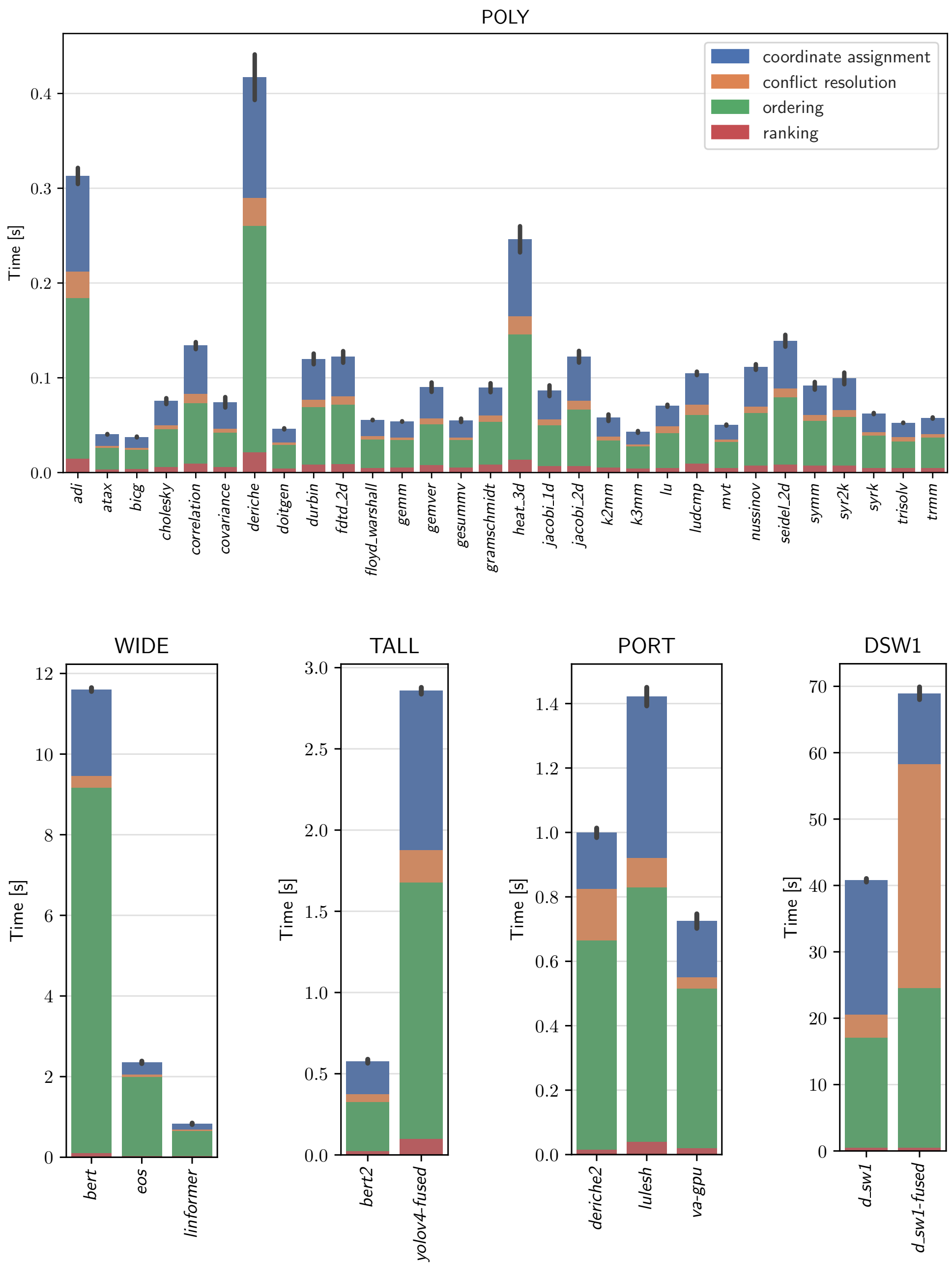


Figure 8.4: Time needed for each step of SUG-J for various graphs. Vertical bars at the top indicate the standard deviation for the total layout time between different runs.

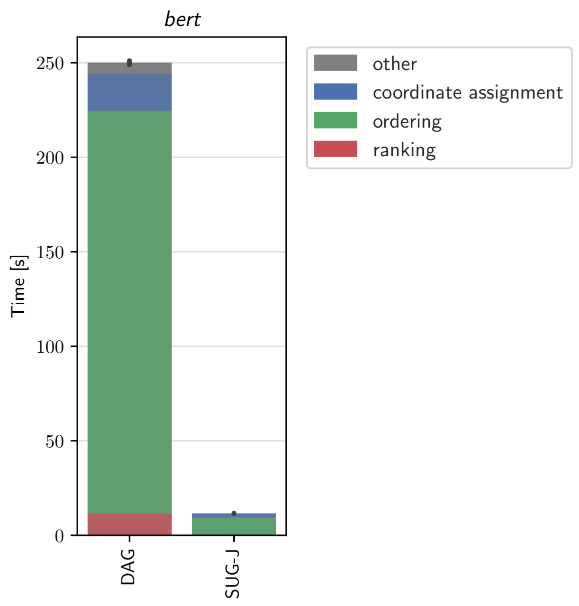


Figure 8.5: Time spent in steps of DAG versus steps of SUG-J for graph *bert*. The time for conflict resolution (that only exists in SUG-J but not in DAG) is included in the 'ordering' step.

already noticeably increases the time needed for conflict resolution; and in graph *d_sw1-fused* with 499 heavy-light conflicts, this step makes for about 49% of the total time. The rest of the time is spent in the ordering and the coordinate assignment step. In most cases, the ordering phase is longer than the coordinate assignment phase because the complexity of the ordering, $\mathcal{O}(S \cdot (|E''| \log |V''|))$, is the dominating factor in the absence of conflicts. As expected, for more parallel graphs as in the WIDE and PORT set, the ratio of ordering to coordinate assignment is larger than for rather sequential graphs as in TALL. The only graph where the coordinate assignment takes longer than the ordering is *d_sw1*, a highly sequential graph.

In Figure 8.5, we show the execution time of different steps in the `Dagre` layouter for graph *bert*. Despite not being the largest graph in our test set, *bert* is well-suited for evaluating the scalability of our algorithms, because it is essentially just one large component. (Recall that we lay out components independently from each other.) When comparing it to our SUG-J implementation, we find that the division of the total time into the single steps is similar in both implementations. However, the ranking step makes for about 4.7% of the total time in DAG but only 0.86% in SUG-J. We attribute this to the lower complexity of our tight tree ranking algorithm compared to the network simplex algorithm employed in `Dagre`. As a consequence, the ranking in DAG takes almost as long as the entire layout process in SUG-J.

Multithreading We parallelized two parts of our Sugiyama layouter: The ordering with random initial permutations (shuffling) and the x-coordinate assignment (Brandes-Köpf).
Figure 8.6 compares the total time needed for the ordering step in the multi-threaded version with the single-threaded one. We use the WIDE set here because the ordering step has been shown to be the bottleneck for this set. We choose the number of threads to match the number of logical cores on our machine, that is 1 main thread + 7 worker threads. We failed to conduct this experiment in Google Chrome as it repeatedly crashed during execution [1]. The results thus stem from Mozilla Firefox (version 92.0.2). The first observation is that – as expected – the time in the sequential execution is roughly proportional to the number of initial permutations. Then, we see that the parallel implementation eventually outperforms the sequential for *bert*, but not for *eos*. There is a significant overhead in starting the ordering routine in a worker; the graph has to be converted to an array that can be passed to the worker and then converted back to the original graph object. When this overhead dominates the time needed for the actual ordering, it is better to do everything in one thread. Note that *bert* is much larger than *eos* and therefore the relative overhead is smaller. A solution could be to invoke the workers only for graphs with some minimum number of nodes. Next, we see that the time for *bert* in the parallel implementation jumps from 6 seconds for no shuffles to 21 seconds for 1 shuffle. Some of this increase comes from our choice to return only the number of crossings from the workers to the main thread; the main thread then has to rerun the ordering phase from the best starting configuration. This could be prevented by a more sophisticated implementation where workers return the resulting graphs together with the crossing number and the best graph is reconstructed. Finally, we observe that the curve between 1 and 7 shuffles the line for the multi-threaded implementation is remarkably flatter than the other but jumps again from 7 to 8 shuffles – at the point where the shuffles can no longer be run completely in parallel on our test machine. Note that we can run 7, but not 8 shuffles truly in parallel on our test machine. This does not exactly match our hypothesis that we can just spawn additional worker threads without additional time. Instead, each additional thread adds a penalty of a few seconds. This may be due to copying data between the main thread and the workers and some additional synchronization overhead. However, for large components the multi-threaded version is considerably faster than the single-threaded one when applying 3 or more shuffles (and for large components).
The second parallelized part is the x-coordinate assignment of nodes. The effect of parallelization is shown in Figure 8.7. The four calls to Algorithm 7

[1]We believe that it runs into some internal memory limit. Choosing a lower number of workers or a test set with smaller graphs results in flawless execution.

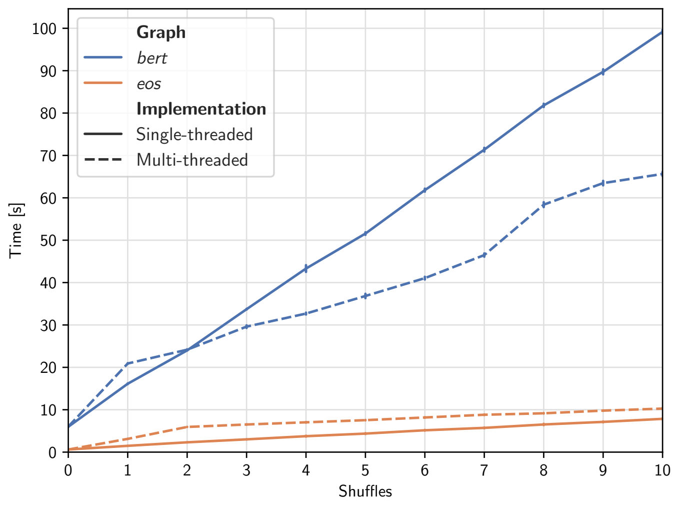


Figure 8.6: Time needed for the entire ordering step with and without worker threads. Here, we use 7 worker threads; together with the main thread this matches the number of hardware threads. Vertical bars indicate the standard deviation of the measurements.

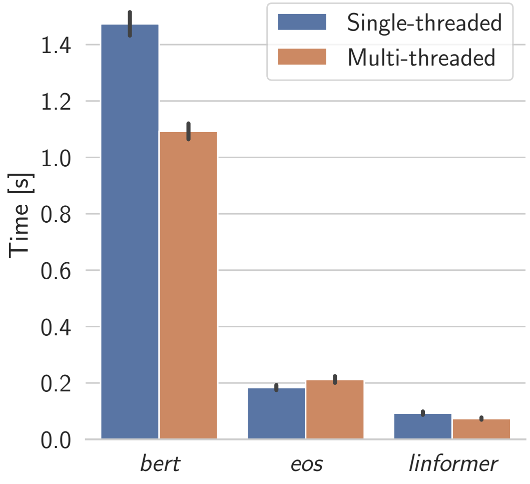


Figure 8.7: Time spent for assigning x-coordinates to nodes with and without worker threads. Vertical bars indicate the standard deviation of the measurements.

are delegated to four worker threads. In contrast to the ordering, this has a positive effect for all graphs of the WIDE set, though the greatest time saving is again found for the *bert* graph.

WebAssembly We next evaluate the capability of WebAssembly (WASM) to improve the performance of our algorithm. Table 8.2 compares the time needed for ordering the nodes of the *bert* graph with different implementations and environments. We have two pure JavaScript versions (JS-v0

Table 8.2: Median runtime in milliseconds for the same procedure (ordering nodes of *bert*) in different environments.

	Chrome		Firefox	
	cold	*warm*	*cold*	*warm*
JS-v0	927	759	578	561
JS-v1	547	211	313	285
WASM	245	166	408	393
Native binary		85		

and JS-v1), a WASM version, and a native binary (compiled with gcc and run directly). Version JS-v1 is a manually optimized version of JS-v0. The improvement primarily comes from two changes: First, v1 tries to reuse allocated memory as much as possible. And second, it uses a faster, in-place sorting algorithm. We test all versions in both Google Chrome (version 91.0.4472) and Mozilla Firefox (version 88.0). Recall that JS engines optimize code just-in-time and hence the performance between contiguous runs may highly differ. To this end, we report here for all browser tests a result for a *cold* run and one for a *warm* run. Cold always refers to the very first run after starting the browser. Warm is the time for the 6th run in a row – at this point the performance has roughly converged.

Our first observation is that performance can differ between different browsers. While JS-v0 is significantly faster in Firefox than in Chrome, the opposite holds for JS-v1 (after warm-up). It becomes also evident that the just-in-time optimization in Chrome's V8 engine is much stronger than the one of Firefox's Gecko engine. We see that in this particular case, the use of WASM yields a small speed-up in Chrome but not in Firefox. One advantage of WASM is that it is already fast in the first run. Note that difference between *cold* and *warm* for WASM comes from the fact that the times stated in the table also include a JS part to restructure the data and pass it to WASM (and the other way round when it is done). As of today, we can only generate 32-bit WASM binaries through Emscripten. We therefore compare it to a native 32-bit binary compiled from the same underlying C code. We see that the execution time of the WASM version is at least 2.5 times that of the native binary (211 vs. 85 ms). Note, however, that the WASM measurement includes the time spent in the JS bridge as mentioned before. We conclude that WASM is able to boost the performance of a web application. For maximum efficiency, one should not switch forth and back between JS and WASM, but port the entire application to WASM. On the other hand, a great deal of improvement can also come from optimizing the JS code – in particular with regard to memory allocation.

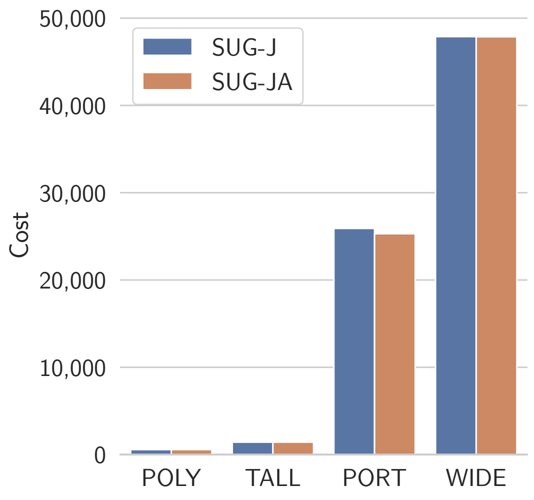


Figure 8.8: Effect of angle optimization on the total cost. The cost is summed over all graphs of the respective set.

Angle Optimization Next, we study the effect of the optional angle optimization step described in Section 5.10. Figure 8.8 shows the total cost for our various test sets with (SUG-JA) and without (SUG-J) angle optimization. We find that the benefit is limited, the largest reduction being observed 2.3% for PORT. One of the reasons for the restricted success might be that crossings usually appear between two ranks that have many edges; then increasing the distance between two ranks quickly increases the cost induced by the edge lengths. We believe the gain could be greater if we did not use a global ranking, as this amplifies the aforementioned effect.

Edge Bundling Table 8.3 shows the number of crossings for graphs of the PORT set with (SUG-JB) and without (SUG-J) edge bundling (see Section 5.11). The bundling is able to cut the number of crossings in the graph *lulesh* greatly. On the other hand, it *increases* the crossings in *deriche2*. This might be due to the fact that we place the port bundles only after minimizing the crossings. Another approach where bundles of ports are regarded as single port during crossing minimization could yield better results. For *va-gpu* the bundling has no effect since there are no edges that could be bundled in the graph.

Table 8.3: Number of edge crossings with and without edge bundling.

Graph	Without bundling	With bundling
deriche2	123	144
lulesh	10366	2957
va-gpu	771	771

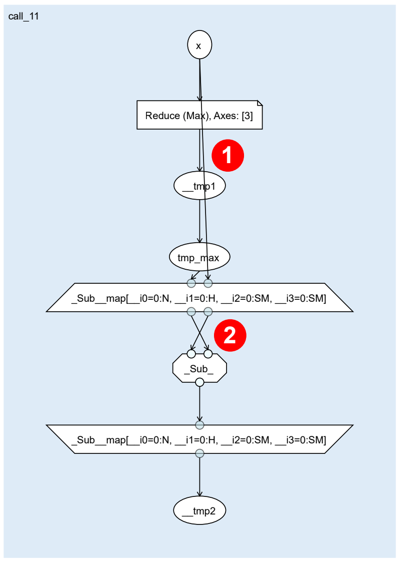

call_11
x
Reduce (Max), Axes: [3]
1
__tmp1
tmp_max
_Sub__map[__i0=0:N, __i1=0:H, __i2=0:SM, __i3=0:SM]
2
Sub
_Sub__map[__i0=0:N, __i1=0:H, __i2=0:SM, __i3=0:SM]
__tmp2

Figure 8.9: Example layout after 1,000 iterations with relatively strong spring force ($w_m = 1$, $w_s = 5$, $w_r = 1$, $\beta = 1$). The example shows an edge overlapping several nodes (1) and poorly ordered ports (2).

8.3 Force-Directed Approach

Regarding the magnetic spring model, we are first interested in the quality of the generated layouts. That is, whether the layouts fulfill our requirements such as all edges going downwards and having no overlaps. To that end, we evaluate the layouter on the POLY set. Results for few sets of parameters that prioritize different goals can be found in Table 8.4. It becomes evident that there are two striking shortcomings in the resulting layouts that persist through all our experiments: First, for virtually all graphs in the test set, the resulting layouts contain node-edge overlaps (see for example Figure 8.9). This does not come as a surprise, since we have a repulsive force only between nodes but not between nodes and edges. Second, the generated layouts are generally too 'loose', that is the drawings tend to be extraneously tall and wide. Setting the spring force relatively high (compared to the other forces) seems to be essential to contain the total drawing area – but at the cost of more violations of our constraints. The relatively high spring force (second line in Table 8.4) still implies a total area that is 55 times larger than that of our Sugiyama-based algorithm.

It is also worth noting that with the magnetic spring layouter, the larger a graph is, the more iterations it takes to turn it from a random initial configuration to an upward drawing. This is due to potential upward paths with many nodes where each node can only assume its final position after its in-neighbor has assumed theirs.

Table 8.4: Magnetic spring layouter applied to the 30 SDFGs of the POLY set. We report for three different constraints how many of the generated layouts violate it. We also indicate the total area required for all graphs together in million pixels (Mpx). The results are for 1,000 iterations.

Parameters	**Upward edges**	**Node-node overlaps**	**Node-edge overlaps**	**Total area [Mpx]**
$w_m = 1$ $w_s = 1$ $w_r = 1$ $\beta = 1$	9/30	6/30	29/30	91,601
$w_m = 1$ $w_s = 5$ $w_r = 1$ $\beta = 1$	6/30	12/30	29/30	5,095
$w_m = 2$ $w_s = 1$ $w_r = 1$ $\beta = 5$	0/30	0/30	30/30	138,258
SUG-J	0/30	0/30	0/30	92

Chapter 9

Conclusion & Future Work

In this thesis, we reduce SDFG layouting to NGP-DAG layouting. We define few hard constraints as well as an additional cost function for rating the quality of generated layouts. We present two alternative solutions to the problem, one based on the Sugiyama framework [68, 29] and one on the magnetic spring model [67]. We evaluate the scalability and quality of our solutions on a set of real-world graphs. We find that the one based on the Sugiyama framework meets our constraints and lays out even the largest graphs in our test set in reasonable time. We conclude that it is well-suited for drawing NGP-DAGs – such as SDFGs – in practice.

Future Work In the magnetic spring model, the magnetic force needs to be relatively strong to ensure downward edges; however, the size of layouts generated under a strong magnetic force are many times higher than the ones generated with the Sugiyama-based layouter. It could be better to set the initial nodes' y-coordinates with the leveled approach. This does not induce much extra time if one uses a fast ranking algorithm, such as our Algorithm 2. The ideal edge length d used in the spring force could be chosen individually for each edge to reflect the graph theoretic distance (as suggested by Kamada and Kawai [44]). One could consider a hybrid layouter that assigns y-coordinates solely based on the Sugiyama solution but the x-coordinates in a force-directed manner. In any case, towards scalability, the complexity of the model has to be reduced, either by existing ideas [3, 27, 25] or new ones.

The complexity of our Sugiyama-based algorithm may also be revisited. Our global ranking scheme results in more compact layouts than the alternative hierarchical ranking; but at the cost of potential conflicts that can take a significant amount of time to resolve. An appropriate treatment of gated child graphs under such a hierarchical ranking is not trivial, but we believe it to be worth exploring. Further, in Section 5.4 we briefly present two options to reduce the number of sweeps S in the ordering phase: Requiring each sweep to remove a certain fraction of the remaining crossings, or imposing a fixed bound. It remains to be examined how each of these alternatives affects performance and quality.

Finally, edge labels are important to understand a graph. This also holds for SDFGs, though the problem is less severe in the SDFV as it can show edge labels when hovering the mouse over the edges. Still, labels are an essential feature in many graph applications. The easiest way to handle them, is to just place each label in the middle of the corresponding edge after the layout is calculated. However, this will likely result in overlaps when the graph is dense. A better approach might be to model labels as separate nodes since we require nodes to not overlap each other.

We believe that after addressing the aforementioned points, our algorithms may not only constitute viable, scalable solutions for drawing NGP-DAGs, but graphs in general.

Bibliography

[1] About emscripten. https://emscripten.org/docs/introducing_emscripten/about_emscripten.html. Accessed: 2021-08-03.

[2] Fv3core. https://github.com/VulcanClimateModeling/fv3core, 2020-2021.

[3] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[4] Wilhelm Barth, Michael Jünger, and Petra Mutzel. Simple and efficient bilayer cross counting. In *International Symposium on Graph Drawing*, pages 130–141. Springer, 2002.

[5] Carlo Batini, Enrico Nardelli, and Roberto Tamassia. A layout algorithm for data flow diagrams. *IEEE Transactions on Software Engineering*, SE-12(4):538–546, 1986.

[6] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.

[7] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[8] Bonnie Berger and Peter W Shor. Approximation alogorithms for the maximum acyclic subgraph problem. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 236–243, 1990.

[9] Therese Biedl and Goos Kant. A better heuristic for orthogonal graph drawings. *Computational Geometry*, 9(3):159–180, 1998.

[10] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[11] Ulrik Brandes and Boris Köpf. Fast and simple horizontal coordinate assignment. In *International Symposium on Graph Drawing*, pages 31–44. Springer, 2001.

[12] Liran Carmel, David Harel, and Yehuda Koren. Combining hierarchy and energy for drawing directed graphs. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):46–57, 2004.

[13] Norishige Chiba, Takao Nishizeki, Shigenobu Abe, and Takao Ozawa. A linear algorithm for embedding planar graphs using pq-trees. *Journal of computer and system sciences*, 30(1):54–76, 1985.

[14] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Hoi-Ming Wong. Upward planarization layout. *Journal of Graph Algorithms and Applications*, 15(1):127–155, 2011.

[15] Vasek Chvatal, Vaclav Chvatal, et al. *Linear programming*. Macmillan, 1983.

[16] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics (TOG)*, 15(4):301–331, 1996.

[17] Emilio Di Giacomo, Walter Didimo, Giuseppe Liotta, and Henk Meijer. Computing radial drawings on the minimum number of circles. In *International Symposium on Graph Drawing*, pages 251–261. Springer, 2004.

[18] Tim Dwyer, Yehuda Koren, and Kim Marriott. Ipsep-cola: An incremental procedure for separation constraint layout of graphs. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):821–828, 2006.

[19] Peter Eades. A heuristic for graph drawing. *Congressus numerantium*, 42:149–160, 1984.

[20] Peter Eades, Xuemin Lin, and W.F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[21] Peter Eades and Kozo Sugiyama. How to draw a directed graph. *Journal of Information Processing*, 13(4), 1990.

[22] Peter Eades and Nicholas C Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994.

[23] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of sugiyama's algorithm for layered graph drawing. In *International Symposium on Graph Drawing*, pages 155–166. Springer, 2004.

[24] Ulrich Fößmeier and Michael Kaufmann. Drawing high degree graphs with low bend numbers. In Franz J. Brandenburg, editor, *Graph Drawing*, pages 254–266, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[25] Yaniv Frishman and Ayellet Tal. Multi-level graph layout on the gpu. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1310–1319, 2007.

[26] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[27] Pawel Gajer, Michael T Goodrich, and Stephen G Kobourov. A multi-dimensional approach to force-directed layouts of large graphs. In *International Symposium on Graph Drawing*, pages 211–221. Springer, 2000.

[28] Emden R Gansner, Yehuda Koren, and Stephen North. Graph drawing by stress majorization. In *International Symposium on Graph Drawing*, pages 239–250. Springer, 2004.

[29] Emden R Gansner, Eleftherios Koutsofios, Stephen C North, and K-P Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[30] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[31] Michael R Garey and David S Johnson. Crossing number is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 4(3):312–316, 1983.

[32] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery.

[33] David Harel and Yehuda Koren. Drawing graphs with non-uniform vertices. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '02, page 157–166, New York, NY, USA, 2002. Association for Computing Machinery.

[34] Patrick Healy and Ago Kuusik. Algorithms for multi-level graph planarity testing and layout. *Theoretical Computer Science*, 320(2-3):331–344, 2004.

[35] Patrick Healy and Nikola Nikolov. *Hierarchical Drawing Algorithms*, pages 409–454. 08 2013.

[36] Patrick Healy and Nikola S Nikolov. How to layer a directed acyclic graph. In *International Symposium on Graph Drawing*, pages 16–30. Springer, 2001.

[37] Patrick Healy and Nikola S Nikolov. A branch-and-cut approach to the directed acyclic graph layering problem. In *International Symposium on Graph Drawing*, pages 98–109. Springer, 2002.

[38] Dion Häfner. Hpc benchmarks for python. https://github.com/dionhaefner/pyhpc-benchmarks, 2019-2021.

[39] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[40] Weidong Huang, Seok-Hee Hong, and Peter Eades. Effects of crossing angles. In *2008 IEEE Pacific Visualization Symposium*, pages 41–46, 2008.

[41] Wolfram Research, Inc. Mathematica, Version 12.3, 2021. Champaign, IL, 2021.

[42] Michael Jünger, Eva K Lee, Petra Mutzel, and Thomas Odenthal. A polyhedral approach to the multi-layer crossing minimization problem. In *International Symposium on Graph Drawing*, pages 13–24. Springer, 1997.

[43] Michael Jünger and Petra Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. In *Graph Algorithms And Applications I*, pages 3–27. World Scientific, 2002.

[44] Tomihisa Kamada, Satoru Kawai, et al. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989.

[45] Roman Klapaukh, David J Pearce, and Stuart Marshall. Towards a vertex and edge label aware force directed layout algorithm. 2014.

[46] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[47] Kurt Koffka. *Principles of Gestalt psychology*. Routledge, 2013.

[48] Joseph B Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[49] Joseph B Kruskal and Judith B Seery. Designing network diagrams. In *Proceedings of the First General Conference on Social Graphics*, pages 22–50. US Department of the Census, 1980.

[50] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. Springer Science & Business Media, 2012.

[51] Kurt Mehlhorn. Graph algorithms and np-completeness, volume 2 of data structures and algorithms. *EATCS Monographs on Theoretical Computer Science. Springer*, 1984.

[52] Kurt Mehlhorn and Petra Mutzel. On the embedding phase of the hopcroft and tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996.

[53] Microsoft. Microsoft automatic graph layout. https://github.com/microsoft/automatic-graph-layout, 2007.

[54] Petra Mutzel. An alternative method to crossing minimization on hierarchical graphs. *SIAM Journal on Optimization*, 11(4):1065–1080, 2001.

[55] Chris Pettitt. dagre - graph layout for javascript. https://github.com/dagrejs/dagre, 2012-2021.

[56] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In *International Symposium on Graph Drawing*, pages 248–261. Springer, 1997.

[57] Helen C Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages & Computing*, 13(5):501–516, 2002.

[58] Helen C Purchase, Robert F Cohen, and Murray James. Validating graph drawing aesthetics. In *International Symposium on Graph Drawing*, pages 435–446. Springer, 1995.

[59] Helen C Purchase, Matthew McGill, Linda Colpoys, and David Carrington. Graph drawing aesthetics and the comprehension of uml class diagrams: an empirical study. In *InVis. au*, pages 129–137. Citeseer, 2001.

[60] Ulf Rüegg, Thorsten Ehlers, Miro Spönemann, and Reinhard von Hanxleden. A generalization of the directed graph layering problem. In *International Symposium on Graph Drawing and Network Visualization*, pages 196–208. Springer, 2016.

[61] Andrew P Sage. Methodology for large-scale systems. 1977.

[62] Georg Sander. Layout of compound directed graphs, 1996.

[63] ETH Zurich Scalable Parallel Computing Lab. Sdfv - the sdfg viewer. `https://github.com/spcl/dace-webclient`, 2019-2021.

[64] ETH Zurich Scalable Parallel Computing Lab. Daceml. `https://github.com/spcl/daceml`, 2020-2021.

[65] Christoph Daniel Schulze, Miro Spönemann, and Reinhard Von Hanxleden. Drawing layered graphs with port constraints. *Journal of Visual Languages & Computing*, 25(2):89–106, 2014.

[66] Kozo Sugiyama and Kazuo Misue. A simple and unified method for drawing graphs: Magnetic-spring algorithm. In *International Symposium on Graph Drawing*, pages 364–375. Springer, 1994.

[67] Kozo Sugiyama and Kazuo Misue. Graph drawing by the magnetic spring model. *Journal of Visual Languages & Computing*, 6(3):217–231, 1995.

[68] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[69] Roberto Tamassia, editor. *Handbook of graph drawing and visualization*. CRC press, 2013.

[70] Robert E Tarjan. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. *Mathematical Programming*, 78(2):169–177, 1997.

[71] J Utech, J Branke, H Schmeck, and Peter Eades. An evolutionary algorithm for drawing directed graphs. In *Proc. of the Int. Conf. on Imaging Science, Systems and Technology*, pages 154–160, 1998.

[72] Frank Van Ham and Bernice Rogowitz. Perceptual organization in user-generated graph layouts. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1333–1339, 2008.

[73] Thorsten Von Eicken, David E Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. *ACM SIGARCH Computer Architecture News*, 20(2):256–266, 1992.

[74] Vance Waddle. Graph layout for displaying data structures. In *International Symposium on Graph Drawing*, pages 241–252. Springer, 2000.

[75] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. Cognitive measurements of graph aesthetics. *Information visualization*, 1(2):103–110, 2002.

[76] John N Warfield. Crossing theory and hierarchy mapping. *IEEE Transactions on Systems, Man, and Cybernetics*, 7(7):505–523, 1977.

[77] Tobias Wicky and Dominic Hofer. dawn2dace. `https://github.com/MeteoSwiss-APN/dawn2dace`, 2019-2021.

[78] Jonathan X Zheng, Samraat Pawar, and Dan FM Goodman. Graph drawing by stochastic gradient descent. *IEEE transactions on visualization and computer graphics*, 25(9):2738–2748, 2018.

[79] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. Npbench: A benchmarking suite for high-performance numpy. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 63–74, New York, NY, USA, 2021. Association for Computing Machinery.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Scalable Drawing of Nested Directed Acyclic Graphs With Gates and Ports

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Leu	Thomas

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Wädenswil, August 20, 2021	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.