

Drawing Graphs within Restricted Area^{*}

Maximilian Aulbach¹, Martin Fink², Julian Schuhmann¹, and Alexander Wolff¹

¹ Lehrstuhl für Informatik I, Universität Würzburg, Germany

² Department of Computer Science, University of California, Santa Barbara, USA

Abstract. We study the problem of selecting a maximum-weight subgraph of a given graph such that the subgraph can be drawn within a prescribed drawing area subject to given non-uniform vertex sizes. We develop and analyze heuristics both for the general (undirected) case and for the use case of (directed) *calculation graphs* which are used to analyze the typical mistakes that high school students make when transforming mathematical expressions in the process of calculating, for example, sums of fractions.

1 Introduction

Our motivation for the problem that we study in this paper stems from so-called *calculation graphs*. Calculation graphs represent calculations starting from some initial task. They are used in studies [13] involving large numbers of high school students in order to analyze the students' typical mistakes in elementary mathematics. Even for relatively simple tasks such as evaluating the term " $3 \cdot (2 + 1/5)$ ", the different transformations performed by a large number of subjects can result in calculation graphs with hundreds of vertices. With the help of drawings of calculation graphs, human experts can analyze how students calculate and, in particular, which mistakes they frequently make. As Hennecke [13] suggests, such drawings are only useful if they are not too large, that is, if they fit into a relatively small drawing area. Hence, well-readable drawings of important parts of the graphs must be generated in an automated fashion. Certainly, the drawn subgraph should represent as much information of the original calculation graph as possible. Therefore, we consider the graphs to be edge- and vertex-weighted; see Fig. 1 for an example. The weight of an edge is the number of students who applied the respective calculation step; the weight of a vertex is the number of students who had the given term as an intermediate result in their calculation. Since we want the labels of the vertices to be readable, we assume that their sizes are fixed. Hence, often only a small fraction of the graph will fit into the prescribed drawing area. Note that a user of such a drawing must be made aware that only a subgraph is shown.

Certainly, vertices and edges that occur only in a single student's calculation can easily be dropped. For higher weights, however, this is not as easy. For instance, it is conceivable that dropping a single vertex of weight W makes it possible to include several other vertices into the drawing, each of weight slightly less than W ; then the resulting drawing potentially allows for a better analysis. Hence, we need to select the subgraph to be drawn based on the graph structure and not only on the weights.

^{*} M. Fink was partially supported by a fellowship within the Postdoc-Program of the German Academic Exchange Service (DAAD). A. Wolff acknowledges support by the ESF EuroGIGA project GraDR (DFG grant Wo 758/5-1).

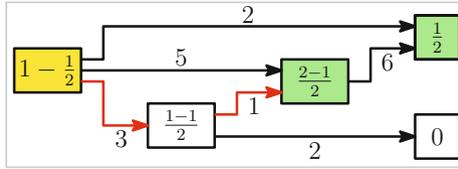


Fig. 1. Example of a calculation graph. Edge weights are indicated on the edges

Further Examples. There are many other scenarios in which most of the information of a large graph should be presented in a limited area. A first example are social networks, which are usually quite large. In this setting, vertices usually have labels and weights; the weight may describe the person's activity in the network. By drawing a heavy subgraph in a restricted area, a user can quickly get an overview over important actors in the network and connections between them. A second example, which we will use in the main part of the paper, are coauthor networks, where the weights represent numbers of publications. Here, we want to find a drawing of a subgraph that represents as many of the publications as possible, i.e., we prefer to keep authors with many publications.

Related Work. Surprisingly, the very natural problem of selecting a subgraph that can be drawn within a prescribed area seems to be new. Some work on related problems, however, is worth being mentioned. Rather than fixing the drawing area and maximizing the size of the subgraph that fits into this area, graph drawing research has focussed on drawing the whole input graph and minimizing the area needed for the drawing, which has also been the task in some graph drawing contests [6]. This problem is called compaction and known to be NP-hard for orthogonal drawings with given orthogonal representation (that is, fixed bends) [14]. In constrained graph layout [12,9], the user can constrain the region into which a vertex must be placed. Dwyer et al. [8] presented the algorithm IPSep-CoLa where constraints demanding a vertical or horizontal separation of vertex labels can be specified. Such constraints make it possible to enforce that vertices are placed within a prescribed rectangular area. However, in contrast to our work, they do not consider dropping vertices. If the vertices do not fit into the area, their algorithm will not find a feasible drawing. The well-known algorithm of Fruchterman and Reingold [10] for force-directed graph drawing allows to specify a rectangular area within which the whole graph has to be drawn. However, in contrast to our work, the algorithm can easily achieve this since vertices are drawn as points without labels and edges can be made arbitrarily short, which we do not allow.

Dwyer et al. [7] developed a method for interactive exploration of graphs based on constrained graph layout. They use a fast heuristic for the overview drawing; for the detailed view of a smaller part of the graph, however, they can afford to use a slower constrained graph layout algorithm that yields better results. Constraints ensure consistency between the views and preserve the users's mental map. Da Lozzo et al. [5] considered the problem of drawing graphs on the very small display of a smartphone. They did not try to represent the whole graph on the display, but rather decided to provide only a local view around a focus vertex; their approach then offers interactive ways of navigating through the graph and exploring the graph based on the local view.

Our Contribution. While calculation graphs have quite specific characteristics and drawing requirements (see Section 3), the problem of graph drawing under area restrictions is also of interest for general, undirected graphs. For both cases, we present heuristics and evaluate them experimentally.

For the general case, we use the force-directed approach, reusing forces defined by Fruchterman and Reingold [10] and by Bertault [3]. We add extra phases in which the graph is pressed together; from time to time we remove vertices or edges from the graph if this is needed for further compaction. We use the desired edge length as parameter both for the usual iterations and our extra phases. In our tests, this proved to be effective for parametrizing the density of the output drawing. Furthermore, we experimentally improved the way of avoiding vertex-edge intersections and we developed a postprocessing that routes some of the edges as curves.

For calculation graphs, we chose the well-known Sugiyama framework [17] as basis for our algorithm since we want calculation paths to be readable from left to right. We add a method that successively removes the least important vertices and edges until the drawing fits into the given area. We also consider the weight of edges for crossing minimization so that important edges have few crossings. In our tests, it turned out that removing the lightest vertices as a preprocessing often improved the weight of the final subgraph. Furthermore, routing the edges as curves gave very nice results, also compared to the original orthogonal drawing style for calculation graphs.

2 General Graphs

We first present an algorithm for drawing arbitrary graphs. The input of our problem consists of an unweighted graph $G = (V, E)$ with a weight function $w: V \cup E \rightarrow \mathbb{R}^+$. For each vertex $v \in V$, we are given a geometric object $\ell(v)$ that will represent the vertex in the drawing. Vertices can be represented by different shapes, e.g., rectangles, disks, or ellipses. We will focus on rectangular vertices which are well-suited for text labels of vertices. We will denote the height and the width of $\ell(v)$ by h_v and w_v , respectively.

In addition to the graph input, we are given an axis-parallel rectangle of height H and width W , the *drawing area*. The task is to find a subgraph $G' = (V', E')$ of G with a *nice* drawing of G' within the given drawing area.

The hard constraints for the drawing are clear: Each vertex $v \in V'$ must be represented by $\ell(v)$, the vertices must not overlap, and each edge must connect its incident vertices. However, it needs to be clarified, what a nice drawing is. In our setting with a restricted drawing area, putting vertices close together can allow us to have more vertices—and thus more weight—in the final drawing. Certainly, a drawing with vertices that are very close is not nice; the same holds for very short edges. We will discuss these criteria and more later in detail.

By a straightforward reduction from SUBSET SUM, where we use height 1 for all vertices and the drawing area, we can easily observe that maximizing the weight of the subgraph that can be drawn in a prescribed area is NP-hard.

2.1 Our Algorithm

Our approach uses the force-directed framework; in this class of algorithms, the drawing is incrementally improved, starting with an arbitrary layout. Each improvement step is done by letting *forces*, defined using physical analogies, move the vertices.

In contrast to usual force-directed algorithms, we have to take both the dimensions of vertices and of the prescribed drawing area into account. Therefore, we add two important ingredients: We try to fit the drawing into a frame of decreasing size, and we remove vertices or edges from the graph in order to make the graph smaller so that the current drawing can fit into the current frame. While, as a first idea, fitting the drawing into the given area could be steered by a force pulling all vertices towards the center of the drawing region, this idea has some drawbacks. Therefore, we will introduce a more advanced approach. Similarly, removing vertices could simply be done by removing the lightest vertex in each step. However, this would take neither the structure of the graph nor the current drawing into account. Hence, we introduce a measure for the *stress* of vertices in the current situation; we will always remove the vertex with the highest stress value. In the following paragraphs, we will detail out our algorithm's individual steps.

Forces. We first define the forces used in our algorithm.

- We reuse existing forces from the algorithm of Fruchterman and Reingold [10]; that is, for any pair $u, v \in V$ of vertices, there is a force $F_r(u, v) = l_{\text{unit}}^2 / (d(u, v)) \cdot \vec{uv}$ on v that repels v from u , where \vec{uv} is the unit vector pointing from u towards v . If the vertices are adjacent, that is, if $uv \in E$, there is an additional force $F_a(u, v) = (d(u, v))^2 / l_{\text{unit}} \cdot \vec{vu}$ that attracts v towards u . Both forces use a factor l_{unit} which describes the desired unit edge length. Since the desired edge length heavily influences the density of the drawing, and, hence, the number and weight of vertices and edges that can be placed within the given drawing area, the choice of l_{unit} is crucial for the results. While we allow the parameter to be set freely, we stress that the value must be chosen carefully, taking the sizes of vertices into account, so that one gets nice output drawings.
- Due to the high density of the input graphs and the given sizes of vertices, it may easily occur that vertices are intersected by nonincident edges, which reduces the readability significantly. As a first step to overcome this problem, we use a force that has been introduced by Bertault in his PrEd algorithm [3]: If an edge $\{u, w\}$ intersects a vertex v in its inner region, that is, close to the center of v , then we let a force $F_e(v, \{u, w\}) = (l_{\text{unit}} - d(v, i_v))^2 \cdot \vec{i_v v}$ repel v from $\{u, w\}$, where the point i_v is the orthogonal projection of v onto the straight-line segment \overline{uw} . Note that this does not guarantee that intersections between vertices and edges are avoided. However, such intersections become less likely; in Sec. 2.2, we will see how their number can be further reduced by routing edges as curves.
- For making the drawing more compact, we introduce a force $F_g(v) = d(v, p_c) \cdot \vec{vp_c}$ that attracts each vertex v to the center p_c of the drawing area. In our experiments it turned out that activating this force in later steps of the algorithm reduces the time for finding a final drawing, but also the quality of the output. Therefore, as a default, the force is only active when computing the very first equilibrium layout.

Handling the Frame. The forces described above yield a functional force-directed algorithm which can be applied for getting an initial layout. Once we have an initial drawing, we initialize the frame F as the bounding box of the drawing. Our algorithm iteratively reduces the size of the frame until it matches the prescribed drawing area.

In each step, we first uniformly reduce the height and the width of F by a small amount. It may happen that some vertices (partially) lie outside of the resulting new frame F' . If this is the case for a vertex v , we just place it at the closest position that lies completely within F' . This operation can result in intersections of vertices. Therefore, we compute a new equilibrium state which hopefully solves the intersections.

In all force-directed iterations in which there is a frame, we will always ensure that no vertex leaves the frame. This is done by cutting off the resulting movement vectors; Fruchterman and Reingold [10] did the same for ensuring that no vertex leaves the drawing area in their algorithm—with the difference that they did not shrink the frame but rather started with a very compressed drawing since in their setting vertices are points that can be arbitrarily close.

If there are intersections after computing a new equilibrium layout, we have a clear indication that the graph is still too large for the current frame and, hence, for the desired drawing area. In this case, and also in some more cases, we will remove vertices or edges as described in the next section.

Removing Vertices and Edges. Our indicator for the necessity of the removal of vertices or edges is, roughly speaking, the density of the drawing. If there are too many vertices in the graph for the current frame, then vertices will come very close. Therefore, we decide to remove a vertex or an edge if the minimum edge length is less than a value l_{adj} or if the minimum distance between two nonadjacent vertices is less than a value l_{nadj} . Note that this includes the case of two intersecting vertices.

Deciding what should be removed is more difficult. Since we want to keep as much weight as possible, the natural idea is to remove the lightest vertex. However, this is often not the best choice—even if it is unambiguous. A more advanced criterion should take also the degree of a vertex into account. The higher the number of neighbors of a vertex is, the more information on the graph is lost by removing the vertex.

Stress Calculation. However, we can still do better: So far, the current drawing is not taken into account, although it yields valuable information about the density of vertices in the vicinity of the vertex that should be removed. Thus, we suggest considering the forces in the last equilibrium layout. Even if the total movement vector of a vertex has length zero, this may actually result from strong forces that try to move the vertex to different directions, e.g., if the vertex is “trapped” between many other vertices.

These considerations lead us to a measure that we call the *pressure* of a vertex. Intuitively, the pressure is the maximum strength of forces in roughly opposite directions that act on a vertex. For formalizing this, we subdivide all force vectors applied on the vertex v into eight octants. For each octant, we sum up the force vector. For $i = 0, \dots, 7$ let l_i be the length of the resulting force vector for octant i . Now, we first build the pressure p_i for octant i by comparing l_i with the force vectors in the three opposite directions, i.e., with l_{i+3} , l_{i+4} , and $l_{i+5} \pmod 8$; see Fig. 2. The pressure then is the maximum over

the pairwise minima, i.e., $p_i = \max \{ \min \{ p_i, p_{i+3} \}, \min \{ p_i, p_{i+4} \}, \min \{ p_i, p_{i+5} \} \}$. The total pressure on v is $p(v) = \max \{ p_0, \dots, p_7 \}$.

Now, we must integrate the weight $w(v)$ and the degree $\text{deg}(v)$ —indicators of the vertex’s resistance against pressure—with the pressure in order to get the *stress* $s(v)$. We do so by setting $s(v) = p(v)/(w(v) \cdot (\text{deg}(v) + c_{\text{deg}}))$. Here, c_{deg} is a small positive constant that ensures that we do not get problems for isolated vertices and that steers our preference for keeping isolated vertices in the drawing. With this definition of the stress, we can always choose the vertex with the highest stress for removal.

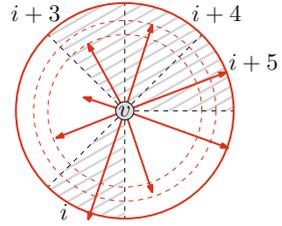


Fig. 2. Pressure computation for one of the octants

Boundary Vertices. There is a special case for vertices close to the boundary of the frame. We never move a vertex over the boundary although, in many cases, vertices are repelled in this direction by the inner vertices. This phenomenon is a cause of pressure on a vertex that is, so far, not covered by our definition. We therefore introduce a new “virtual” force that mimics the resistance against movements that push the vertex out of the frame. This force repels the vertex perpendicularly to the inside of the frame, away from the closest point p_f of the frame’s boundary. More precisely, the virtual force is $F_f(v) = \overline{p_f} \vec{v} \cdot l_{\text{unit}}^2 / d(v, p_f)$. We stress that F_f is only taken into account for the stress computation and not actually applied.

Edges. In some cases, one may try to remove only an edge instead of a whole vertex; the hope is that after removing the edge, the graph becomes more flexible so that the available space can be used better. As an indicator for such a situation, we use the average edge length in the current drawing. If this length is larger than $l_{\text{unit}} \cdot c_{\text{len}}$, with a factor $c_{\text{len}} > 0$, we decide for removing an edge instead of a vertex. The intuition is that, on average, the edges are not very short, which means that by removing one of these longer edges we could allow more flexibility to the placement of vertices.

In order to determine which edge will be removed, we, again, use a definition of stress. To this end, we take both the weight $w(e)$ and the weights of the edges crossing e into account. Let E' be the set of these edges. Then, we set $s(e) = \sum_{e' \in E'} w(e') \cdot |E'| / w(e)$ to be the stress of e , and we remove the edge with the highest stress value.

2.2 Extensions

We developed and implemented two extensions that can help improve the runtime of the algorithm and the quality of the resulting drawings, respectively.

Preprocessing. In many input instances, there is a large number of vertices with very small weight, for which it is very unlikely to occur in the final drawing. To speed up the algorithm, we can remove all vertices that are lighter than a threshold value w^* . Our choice of w^* is based on guessing a bound on the maximum number of vertices in the final drawing and depends on the height H and the weight W of the drawing area, the minimum height h_{min} and the minimum width w_{min} of a vertex as well as on the desired edge length l_{unit} and a factor $c_{\text{pre}} > 0$. We will make sure that we keep at least $(H \cdot W) / ((l_{\text{unit}} c_{\text{pre}} + h_{\text{min}}) \cdot (l_{\text{unit}} c_{\text{pre}} + w_{\text{min}}))$ vertices in the graph.

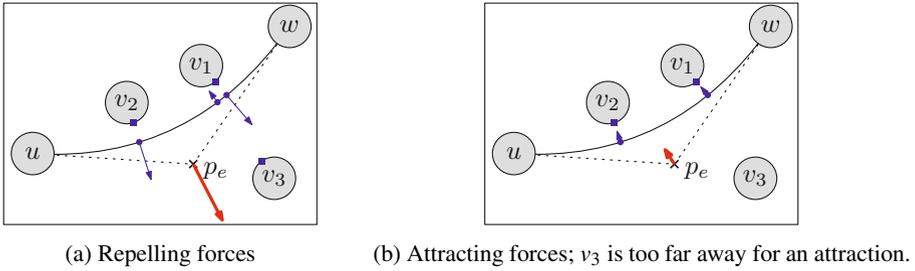


Fig. 3. Repelling and attracting forces for Bézier curves

Postprocessing: Bézier Curves. In Sec. 2.1, we explained the force that aims at avoiding intersections between vertices and nonincident edges. However, we cannot guarantee that we do not have such intersections. We can do two things about this: (i) Intersections of an edge with the outer region of a vertex are relatively easy to distinguish from incidences and can, therefore, be tolerated. (ii) We can remove more edges if necessary.

Here, we present a third possibility: If we allow edges to be curves rather than only straight-line segments, we can avoid more intersections. In their improvement to Bertault’s PrEd algorithm, Simonetto et al. [16] allowed polyline edges, where bends are introduced and removed based on the current drawing. However, this approach made the algorithm much slower; also, use a postprocessing in which edges are routed as Bézier curves around intersected nonadjacent vertices. We do this by representing edge $e = \{u, w\}$ by a quadratic Bézier curve, i.e., a parametric curve with a control point p_e in addition to the endpoints.

The computation of the curve is done as a postprocessing in the very last step of the algorithm. It is realized as an additional force-directed algorithm in which only the control point is moved, starting at the position in the middle between u and w . Each vertex v that is not far away from e causes a repelling force on p_e ; see Fig. 3a. This force is parametrized by the width w_v and the height h_v of v as well as by the point p'_v of v that is closest to e and the point p'_e of e that is closest to v . The repelling force is defined as $F_{rb}(e, v) = \overrightarrow{p'_v p'_e} \cdot (w_v^2 + h_v^2) / d(p'_v, p'_e)$.

In order to avoid that the edge is curved too much, we also have an attracting force for vertices that have been (almost) intersected by e . If v is such a vertex, then the attracting force $F_{ab}(e, v) = \overrightarrow{p'_e p'_v} \cdot d(p'_v, p'_e)^2 / \sqrt{w_v^2 + h_v^2}$ is applied to p_e ; see Fig. 3b.

2.3 Experiments and Evaluation

We implemented our heuristic in Java, using the graph library JUNG[1]. Our experiments were performed on a Core i5-2500K CPU with 8 GB of RAM. For the force-directed part of the algorithm, we also used a cooling factor that slows down the movement of vertices over the iterations in order to accelerate the computation of an equilibrium. We configured our algorithm such that there are always 25 steps of shrinking the frame around the current drawing. As input data, we primarily used (subgraphs of) the graph drawing collaboration graph from 1994 till 2012; in total, the graph has 950 vertices and 2559 edges. The weight of each vertex is the number of publications of

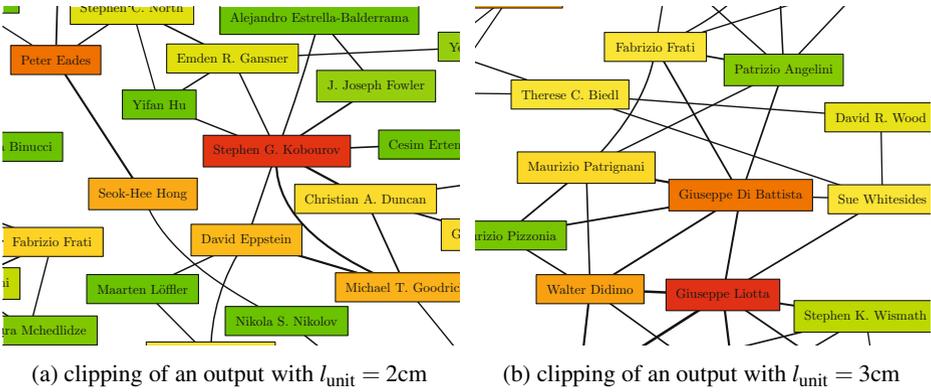


Fig. 4. Output examples with different choices for l_{unit}

the respective author and the weight of an edge connecting two authors is the number of their joint publications. We focused on a drawing area of $29.7\text{cm} \times 21\text{cm}$ (DIN A4) where the size of vertices was determined by the author's name (in 10pt font). For the complete publication graph, the runtimes were about 2 minutes, depending on the parameters.

The main results of our tests are the following.

- The preprocessing step described in Sec. 2.2 pays off; depending on the parameter c_{pre} , we may save runtime and get better results. In our tests, $c_{\text{pre}} = 0.7$ seemed a good choice; see Table 1 in the extended version of the paper [2].
- For testing whether something has to be removed, we set $l_{\text{adj}} = 0.1l_{\text{unit}}$ and $l_{\text{nadj}} = 0.15l_{\text{unit}}$. For the parameter c_{len} that determines whether an edge or a vertex will be removed, $c_{\text{len}} = 0.9$ gave a good compromise between the vertex and the edge weight in the final drawing; see also Table 2 in the extended version [2].
- We tested the effect of not always activating the force that repels vertices from edges; instead, we first computed an equilibrium without the force and then another one with it, so that we have more flexibility for vertices to cross edges. In our tests, this proved to have a significant impact on the total weight of edges in the final drawing, yielding an increase of almost 80%; see Table 3 in the extended version [2].
- The following way of computing the total force vector F for a vertex based on putting weights to the single forces showed the best results: $F = 0.01F_r + 0.01F_a + 0.005F_g + 0.0075F_e + 0.01F_f$. Note that not all forces are active at the same time.

A central parameter used in our algorithm is l_{unit} , describing the desired edge length. Figure 4 shows output examples that demonstrate that higher values lead to drawings that are less dense. Full outputs are available in the extended version of the paper [2]

3 Calculation Graphs

We now consider the initial problem of drawing calculation graphs in prescribed area. In our terminology, most of the input stays the same compared to general graphs, except

that now our edges are directed since they represent calculation steps. Additionally, we are given a *start vertex* $s \in V$ that represents the task given to the students. Hence, s must be present in the output drawing and, furthermore, we insist that in the subgraph that is finally drawn, each vertex can be reached from s . To further improve the readability of the steps, we want as many edges as possible pointing from left to right. As a drawing convention for the edges, we will use the orthogonal drawing style with edges leaving vertices horizontally; we try to minimize the numbers of crossings and of bends and to optimize the gaps between edges and vertices. The hardness proof for general graphs can easily be adjusted. Hence, the problem remains NP-hard.

3.1 Our Algorithm

Due to the required readability from left to right, our approach is based on the Sugiyama framework [17] for hierarchical graph drawing. This framework consists of several steps that we will adjust to our problem. Our adjustments are optimized for our drawing style and especially for the task of removing vertices or edges from the graph, if necessary. We first briefly review the steps of the Sugiyama framework before going into detail and describing our adjustments.

In the first step, the graph is made acyclic by reverting some of the edges. Next, the vertices are assigned to layers from left to right. Then, based on the layer assignment, the number of edge crossings is minimized resulting in relative orders of the vertices of each layer. Eventually, final vertex coordinates are computed and the edges are routed.

Breaking the Cycles. In later steps of the Sugiyama framework, it is assumed that all edges are directed from left to right, i.e., the graph must be acyclic. Hence, we must revert some edges. For reverting the smallest number of edges, the NP-hard Feedback Set problem must be solved. We can either use existing heuristics, or even afford solving the problem optimally with the help of an MIP solver; in our tests, this worked quite fast and allows us also to minimize the weight of the reverted edges rather than their number.

Layer Assignment. Several approaches for the layer assignment in the Sugiyama framework exist, depending on the objective, e.g., of minimizing the number of layers or the number of vertices in a layer. Often, the number of layers is minimized subject to a prescribed maximum height of each layer. However, we will not use the height of our drawing area as the maximum height of a layer, although, at first, this may seem a good idea: If we do so, we would most probably have to remove many layers of vertices completely from the graph in later steps, which, subsequently, can also cause the removal of vertices of layers that are not removed, making these layers automatically smaller. Instead, we will set $n_{\max} = \lceil |V|/k \rceil$, where k is the length of the longest path in the graph, so that we can hope for roughly equal numbers of vertices per layer. We mainly used the heuristic of Coffman and Graham [4] with the minor adjustment that preferably the left-most layers have more vertices; we also tested Graham's list scheduling algorithm [11] and an assignment with the minimum number of layers.

Vertex Removal. After the layer assignment, the configuration usually does not fit into the drawing area. We now remove vertices until all vertices can be placed in the drawing

area. We first remove vertices from each layer, so that the height of the layer is small enough. Afterwards, we remove whole layers until the width requirement is fulfilled. We first remove single vertices because this step can significantly influence the total weight of layers and, therefore, the choice of layers that will be deleted. The removal from the layers is done from left to right since the removal of a vertex from a layer can cause other vertices right of it to also be removed, if they become unreachable from s .

When removing from a layer, we should prefer light vertices. However, we must also take the heights of vertices into account: Removing a high vertex may save as much space as removing several lighter vertices whose weight sums up to a larger value. Hence, we measure the importance of a vertex v as $i(v) = w(v)/h_v$ and remove as many vertices of lowest importance as necessary so that the layer fits into the drawing area. We also tried other importance measures by taking the possible decrease of the width of the layer or the decrease of its area into account when removing a vertex. However, these measures did not perform better than the simpler height-based measure.

Once all layers have a feasible height, we will remove complete layers so that we are within the allowed total width. Note that we must always keep gaps between adjacent layers so that the edges can be drawn. The removal of layers is also done from left to right. We do this based on the importance of a layer L , which we define as $i(L) = \sum_{v \in L} w(v) / \text{width}(L)$, where the width of L is determined by its widest vertex.

Crossing Minimization. For the crossing minimization, we use the methods commonly used in the Sugiyama framework, which are based on considering only (parts of) edges between adjacent layers, but do so multiple times. For the adjacent exchange heuristic, we also considered the version where the weight of crossing edges is minimized. This heuristic just performs swaps of adjacent vertices in a layer—if this reduces the number (or weight) of crossings. Hence, weights can easily be integrated.

Edge Removal. Even after crossing minimization, there could still be too many crossings for the drawing to be well readable, if the graph is dense. Hence, we add a step in which edges are removed, if necessary. To this end, we introduce a measure for the importance of an edge e : If $E'(e)$ is the set of edges that cross e , then the importance of e is $i(e) = w(e) / (\sum_{e' \in E'(e)} w(e'))$. The result is that edges without a crossing are considered most important and will never be removed. Furthermore, an edge that crosses heavy edges—which are more valuable to us—will more likely be removed.

Gaps in Layers. Now, we know the orders of vertices in layers, where a layer also contains edges that are routed through several layers. We improve readability by using different gaps between the objects in a layer: Two edges are drawn closer together than two vertices, and edges that stay parallel until the next layer can be drawn even closer. While the different gaps make the drawings nicer, the consequence is also that only now we know the precise height each a layer. In some cases, this can make it necessary to remove another vertex of a layer, which, again, is done based on the importance measure.

Coordinate Assignment. For the final adjustment of the vertex positions, we still have some flexibility if the vertices (and edges) in the layer do not consume the total available height. We can use this flexibility and try to minimize the number of edge bends.

Therefore, we integrated a part of the heuristic of Sander [15]. However, due to the height constraint, we usually cannot save too many bends.

Edge Routing. Finally, only the edges need to be drawn, with several subproblems:

- We have to distribute the ports of the edges at the incident vertices or use a single port shared by all edges.
- We indicate the weight of edges by drawing them with different width. Since there are only few edges that are very heavy, it makes sense to not use a linear dependency between width and height but, e.g., a logarithmic dependency, or a dependency to the cube root (which gave the nicest results for our drawings).
- We have to distribute the vertical segments of the edges between consecutive layers such that both overlaps between segments and unnecessary (double) crossings are avoided. We first find a relative order of the segments from left to right for each pair of adjacent layers as follows: For any pair of edges between the layers that do not have to cross, there is at most one order with an unnecessary crossing. Using these orders, we build a directed graph of vertical segments. A topological sorting of this graph yields an order of the segments that avoids all unnecessary crossings. Once a relative order of the vertical segments is found, we assign the final coordinates. Small improvements are possible that locally optimize the spacing between the edge segments and we have put a lot of effort into implementing some of them. The most valuable optimization was using a force-directed algorithm with repelling forces between adjacent segments that optimizes the distances between the segments.

3.2 Extensions

Preprocessing. Similar to our algorithm for general graphs, we use a preprocessing step in which very light vertices and edges are removed in order to speed up the later steps.

Reinsertion of Removed Vertices. After crossing minimization, when the order of vertices is fixed, it is possible that we could safely reinsert some of the removed vertices so that the available area is used in a better way. We prefer the vertices with the highest importance as defined before and insert them in the leftmost available layer. Note that after reinserting vertices, we may have to reorder some of layers for crossing minimization.

Bézier Curves. While we try to avoid bends of the orthogonal edges, there still can be longer edges that have several bends, making them hard to follow. We suggest drawing the edges as smooth curves instead. To this end, we represent each edge segment between two adjacent layers as a cubic Bézier curve. As a simple version, this can be done by making the two bends of the orthogonal edge the two middle control points of the curve; this yields already quite nice results. We can further improve the drawings by adjusting the force-directed algorithm for the vertical segments (i.e., the control points): First, we can allow horizontal segments to overlap since they are not actual segments any more. Second, we can add a tendency to put the segments close to the middle between the layers in order to avoid sharp bends. We can, however, not place all vertical segments in the middle; doing so could result in unnecessary crossings of the respective curves.

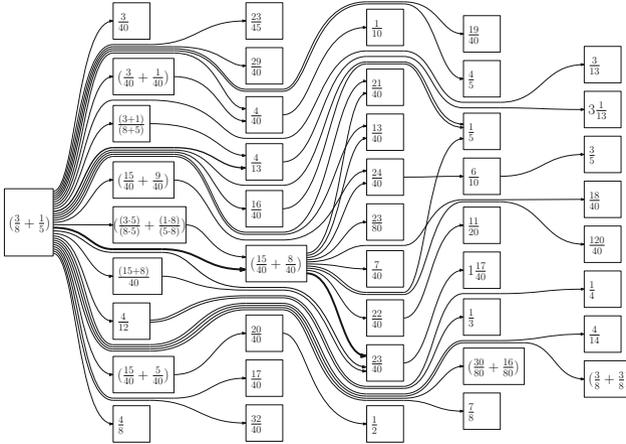


Fig. 5. An output example with edges drawn as Bézier curves (scaled down)

Weight Transfer. Suppose we delete a vertex v such that the edges (u, v) and (v, w) for vertices u and w exist. If both edges are heavy, it is possible that many students reached w from u with v as an intermediate step. Hence, after the removal, an edge (u, w) becomes more valuable to us because this edge can also partially represent the steps described above. We can model this by creating edge (u, w) —if it did not exist—and increasing its weight by $\min \{w(u, v), w(v, w)\}$ for the remainder of the algorithm. Similar weight transfers make sense also in more complicated situations.

3.3 Experiments and Evaluation

Also the algorithm for calculation graphs was implemented in Java. We used real-world data generated in user studies, with graphs of 107 and more vertices. The largest graph had 1031 vertices and 1549 edges. As for general graphs, we mainly used the A4 paper size as the prescribed drawing area. The tests were performed on a 3 GHz CPU with 4 GB RAM. Figure 5 shows an output example using Bézier curves, which, in our opinion, is the nicer and more interesting style compared to the version with orthogonal edges. Full examples can be found in the extended version [2]. Our main results are as follows.

- A preprocessing that removes the lightest vertices often improves the output, i.e., the drawn subgraph is heavier; see Table 4 in the extended version [2].
- There was no significant influence of the chosen layering algorithm on the weight of the final subgraph, especially when using the postprocessing for vertex reinsertion.
- Taking the weight of crossing edges into account in the adjacent exchange heuristic for crossing minimization reduces the weight of crossing edges significantly (factor > 2) and causes only few additional crossings; see Table 5 (extended version [2]).
- The computation for the largest graph with 1031 vertices took 3 to 4 seconds, depending on the parameters; most of the time was spent on the adjustment of segments in the edge routing step and on writing the output file (about 2 seconds).

4 Conclusion

We have introduced the problem of drawing a heavy subgraph in a prescribed area. Both for general graphs without further constraints and for calculation graphs, we have developed and tested heuristics which yield quite nice results.

Acknowledgement. We thank Martin Hennecke for introducing the problem of drawing calculation graphs to us and for providing us with input data.

References

1. Java Universal Network/Graph Framework (JUNG), <http://www.jung.sourceforge.net>
2. Aulbach, M., Fink, M., Schuhmann, J., Wolff, A.: Drawing graphs within restricted area. CoRR (2014), ArXiv e-print <http://arxiv.org/abs/1409.0499>
3. Bertault, F.: A force-directed algorithm that preserves edge crossing properties. Inf. Proc. Letters 74(1-2), 7–13 (2000)
4. Coffman, E.G., Graham, R.L.: Optimal scheduling for two-processor systems. Acta Inform. 1(3), 200–213 (1972)
5. Da Lozzo, G., Di Battista, G., Ingrassia, F.: Drawing graphs on a smartphone. J. Graph Algorithms Appl. 16(1), 109–126 (2012)
6. Duncan, C.A., Gutwenger, C., Nachmanson, L., Sander, G.: Graph drawing contest report. In: Didimo, W., Patrignani, M. (eds.) GD 2012. LNCS, vol. 7704, pp. 575–579. Springer, Heidelberg (2013)
7. Dwyer, T., Marriott, K., Schreiber, F., Stuckey, P., Woodward, M., Wybrow, M.: Exploration of networks using overview+detail with constraint-based cooperative layout. IEEE Trans. Vis. Comput. Graph. 14(6), 1293–1300 (2008)
8. Dwyer, T., Koren, Y., Marriott, K.: IPSep-CoLa: An incremental procedure for separation constraint layout of graphs. IEEE Trans. Vis. Comput. Graph. 12(5), 821–828 (2006)
9. Dwyer, T., Marriott, K., Wybrow, M.: Topology preserving constrained graph layout. In: Tollis, I.G., Patrignani, M. (eds.) GD 2008. LNCS, vol. 5417, pp. 230–241. Springer, Heidelberg (2009)
10. Fruchterman, T.M.J., Reingold, E.M.: Graph drawing by force-directed placement. Softw. Pract. Exper. 21(11), 1129–1164 (1991)
11. Graham, R.L.: Bounds for certain multiprocessing anomalies. Bell Syst. Tech. J. 45(9), 1563–1581 (1966)
12. He, W., Marriott, K.: Constrained graph layout. Constraints 3(4), 289–314 (1998)
13. Hennecke, M.: Rechengraphen. Math. Didact. 30(1), 68–96 (2007)
14. Patrignani, M.: On the complexity of orthogonal compaction. Comput. Geom. Theory Appl. 19(1), 47–67 (2001)
15. Sander, G.: A fast heuristic for hierarchical Manhattan layout. In: Brandenburg, F.J. (ed.) GD 1995. LNCS, vol. 1027, pp. 447–458. Springer, Heidelberg (1996)
16. Simonetto, P., Archambault, D., Auber, D., Bourqui, R.: ImPrEd: An improved force-directed algorithm that prevents nodes from crossing edges. Comput. Graphics Forum 30(3), 1071–1080 (2011)
17. Sugiyama, K., Tagawa, S., Toda, M.: Methods for visual understanding of hierarchical system structures. IEEE Trans. Syst. Man Cyber. 11(2), 109–125 (1981)