Optimum Layout Adjustment Supporting Ordering Constraints in Graph-Like Diagram Drawing

Kārlis FREIVALDS and Paulis ĶIKUSTS

Institute of Mathematics and Computer Science
University of Latvia, 29 Rainis blvd., Riga, Latvia
{karlisf,paulis}@cclu.lv

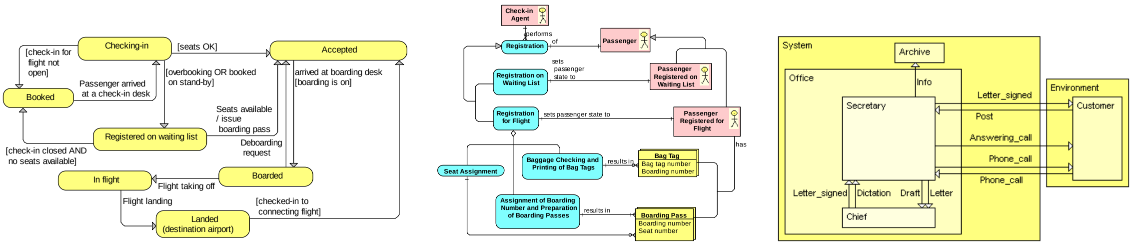
Abstract. We propose an optimization-based technique for layout operations ensuring flexible and convenient interactive editing of a wide class of graph-like diagrams. Diagrams may contain nested nodes, textual labels on connection paths, and branched structures of paths. Layout operations rely on mental map preserving optimum layout adjustment via solving quadratic programming problems subject to ordering constraints.

Introduction

Graph-like diagrams are graph based pictorial models that indicate the interrelationships of elements of various structures. Graph-like diagrams are widely used to describe the information and its structure in such areas as CASE or CAD, for example describing the connections between enterprises, development of specifications, or for program code representation [MM88, BRJ99].

An important aspect for users is diagram visualization, a process called *diagram layout*. Layout technique of graph-like diagrams has been developed hand in hand with pure graph layout [BNT86, RDMMST87, SM81, TDBT88], gradually refining requirements for diagram layouts [DM90, PSTS91]. However, as emphasized in [LE95], pure graph layout on the whole has received more attention [DETT94, DETT99] than diagram layout. In fact, additional requirements for layout of diagrams cause specific problems that could be far from principal questions of pure graph layout. Striking examples are tree-like structures with edge drawing conventions such as combination of several edges into branched fork-like paths, or representation of an edge by geometrical inclusion of node symbols [LE95] (see also [MM88, SM91]). Of course, when it is too tedious to maintain specific requirements, we could ignore them. For example, in [S97], a well-known graph drawing algorithm is used for UML class diagrams [BRJ99], however the inherent UML fork tradition, which has no generally adopted realization in pure graph layout, is simply rejected.

Our graph-like diagrams are combinatorial structures consisting of elements of three principal kinds: nodes, relations among nodes, and labels. A layout of a diagram is an arrangement of geometrical objects on the plane corresponding to the diagram elements (Fig 1).



a **b** **c**

Figure 1. Simple diagram (a), diagram with forks (b), diagram with nested nodes (c).

Nodes are basically represented by two-dimensional symbols, most commonly by upright rectangular boxes or circles. In this paper we will use only rectangular boxes.

Relations are represented by
(1) *paths*, i.e. single rectilinear polylines connecting symbols that represent the associated nodes (Fig. 1a),
(2) *forks*, i.e. branched structures of rectilinear polylines (Fig. 1b),
(3) *inclusions*, i.e. placement of one node symbol inside another one (Fig. 1c).

Labels are text fields, represented by upright rectangles and are categorized into node labels and path labels. Node labels are placed inside the nodes on the specially assigned margins. Path labels must be placed near their lines in an understandable way which label belongs to which line.

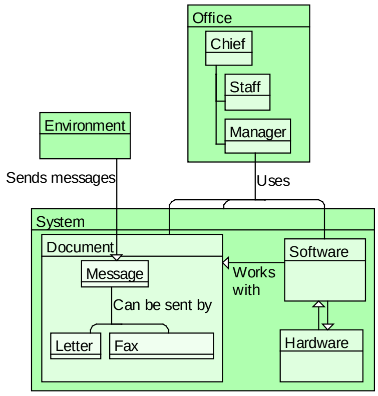


Figure 2. Complex diagram.

Since we allow a wide range of geometrical representations of relations, our layouts cover the full spectrum from drawings of simple graphs (Fig. 1a) up to UML diagrams [BRJ99, SBKP98] (Fig. 1b), including essentially generalized K.Sugiyama's and K.Misue's compound graphs [SM91] (Fig. 1c). Figure 2 shows that all together.

The task for diagram layout is to represent the information of diagrams in an easily perceptible way [DM90, PSTS91]. Accordingly, a correct layout must satisfy natural geometric constraints:
(C1) node rectangles are not smaller than a minimum size,
(C2) path lines have no common segments,
(C3) the minimum distance $\delta > 0$ is guaranteed between nonintersecting segments of geometrical objects,
(C4) path labels neither overlap each other, nor node contours, nor path lines,
(C5) node contours do not cross each other,
(C6) path lines do not intersect node contours unless forced by inclusions.

We allow variable size node rectangles in order to be able to draw graphs of degree higher than four [DETT99, MHT93]. Also editing node labels or putting one node inside another one could cause to change node sizes. Similarly, inserting new nodes or path labels between the paths contacting the same node may require changing its size.

A layout of a diagram can be created interactively by the user, or automatically by a program. The interactive drawing approach [DETT99] has led to the idea of a *mental map* [BT98, DETT99, MELS95]. The mental map of a diagram should be preserved during the layout process in order to ensure the user's control and understanding. Thereby all changes to the diagram have always to be minimized, so optimization approach rises in a natural way along with the notion of mental map.

The concept of a mental map together with optimization questions is profoundly studied in literature. In [MELS95], the problem of preservation of the mental map is discussed. The authors propose several models to make the concept of the mental map more precise: orthogonal ordering, proximity relations, and topology. Additionally an algorithm for rearranging a diagram to avoid node overlap preserving orthogonal ordering is presented. [HIMF98] develops this approach further to avoid intersections among rectangles. In [BT98] a formalization of the notion of mental map is performed, and differences between layouts in various aspects: distance, proximity, orthogonal ordering, shape, and topology are expressed mathematically. The authors of [HM97] use mathematical programming including quadratic one to preserve the mental map in an interactive layout when repeated modifying occurs. Constraints express semantic information, mainly about various aesthetics that have to be considered automatically.

Our approach to the drawing of graph-like diagrams grows from the tool GRADE [KR96], and is now being developed further for Editor Factory needs [SBKP98]. Editor Factory is an annotation language interpreter, which can be used to design various diagram editors. Editor Factory is based on Graphical Diagramming Engine [G], which provides the graphical functionality of the editor and its user interface.

Editors must be able to manage diagrams interactively, and to generate the layout automatically. An editor based on our Graphical Diagramming Engine provides fully automatic layout and direct manual painting of graphical primitives as extreme cases as well as intermediate editing levels all integrated in a single system. Fast procedures for switching among the various layout modes have also been implemented, thus ensuring flexible and convenient editing by filling the gap between the extreme levels of editing. We have to deal with large diagrams consisting of hundreds or even thousands nodes and relations in real time. Therefore the diagram operations must be designed for maximum speed.

To create a layout of a diagram we go through several relatively independent stages. The main of them are node layout, path routing, layout compaction, and label assignment. At each stage we provide a correct intermediate layout trying to preserve a common mental map. To guarantee maintaining the mental map during layout modification we solve two quadratic optimization problems, one in each orthogonal direction.

This approach conforms to several important ideas proposed in the literature. First, layout adjustment requires the objects to move or to stretch as in [MHT93]. Furthermore, an optimum adjustment involves mathematical programming including integer, linear, and quadratic ones that are widely used in graph drawing [HM97, DETT99]. Recent works [BDPP99, KM99] also elaborate related concepts and touch ours in some basic points.

The deviation of the layout from the intended mental map can be measured by a function to be minimized. Our function comes from the idea of distance metrics [BT98] and position constraints [DETT99]. Minimization is done subject to *ordering constraints*. In [DETT99] it is shown how ordering constraints can be used in layered drawings for horizontal coordinate assignment. However when discussing the use of a quadratic programming approach, the authors warn that the solution requires considerable computational resources even if the ordering constraints form an acyclic graph. Such constraints also appear in [GKNV93] for finding optimum layering by integer programming. Below we show that our technique, which is based on the projective gradient method [M89], allows us to solve these problems spending quite moderate computational resources.

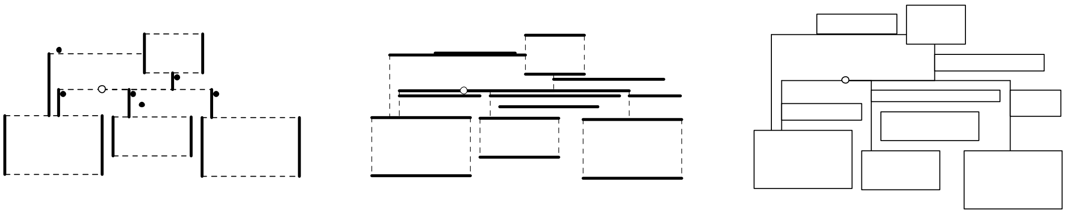
As another example, we have a possibility to eliminate the intersections among upright rectangles while preserving the orthogonal ordering. For this purpose [MELS95, HIMF98] offer an $O(n^2)$-time heuristic algorithm that minimizes the layout area. Our approach gives similar results but in a conceptually easier way. Further, our operations include also other rectangle processing algorithms for rectangle compaction and packing.

A quadratic programming algorithm is the principal part of a procedure called *Normalize*, which ensures a correct intermediate layout while not destroying the common mental map at each layout creation stage. *Normalize* is our backbone operation and is discussed below in more detail.

Layout structure and normalization

When modifying the diagram, the user is inserting new nodes or paths, adding path labels or changing geometrical attributes of diagram elements. Without difficulty all these actions can be accomplished satisfying the constraints C4, C5, and C6, while the other constraints may be violated. To satisfy all our constraints C1…C6 layout *normalization* is needed. Besides, the initial mental map must be preserved.

To satisfy the constraints C4, C5, and C6, new nodes and path labels have to be represented by zero-size rectangles (i.e. points) located in the desired positions (the bold dots in Fig. 3a). As an independent point-shaped object we allow to define also the so-called *support*. A support is the common endpoint of the paths forming a fork (the circle in Fig. 3). Finding a proper position of point-shaped objects is a separate task different for nodes, labels or supports. The position may be pointed out by the user or calculated automatically by the layout algorithm.



a b c

Figure 3. Vertical layout segments (a), horizontal layout segments (b), and all layout segments (c).

Since our diagram elements are represented by upright rectangles or rectilinear polylines, the layout geometry consists only of vertical and horizontal line segments (possibly of zero length because of point-shaped objects) (Fig. 3a). Overlapping paths segments contacting supports are merged during normalization. The *Normalize* operation ensures the constraints C1…C6 including minimal distances between newly routed paths, and minimal sizes of elements while minimizing changes to the diagram. For newly added nodes and path labels *Normalize* assigns correct sizes to these elements. Also correct node inclusions are ensured.

Preservation of the mental map for us means minimization of the total distance between the new and the old places of the diagram elements while keeping their ordering undisturbed. Rectilinearity of the diagram elements allows us to process the total distance and ordering separately in horizontal and vertical directions: first for the vertical layout segments, then for the horizontal layout segments (Figs. 3a, 3b).

Note that after processing the vertical segments, the nodes and path labels represented by points become horizontal segments due to minimum size requirements (Fig. 3b).

Let us consider more closely the case of vertical line segments. In this case we have to find only the x-coordinate of each segment. The objective is to assign x-coordinates to the vertical segments in a way that the chosen cost function attains its minimum and the constraints are taken into account. The basic constraints are minimum horizontal distance requirements.

To keep the general view of the given layout unchanged, the ordering of segments is predetermined in some sense. The main idea here is a segment *obstacle* relation, which is derived from segment visibility: segment b is an obstacle for segment a if

- projections of the extended segments of a and b on the vertical axis overlap,
- the abscissa of a is smaller than the abscissa of b,
- there is no segment c between a and b such that c is obstacle for a, and b is obstacle for c.

Here for the given segment the *extended segment* is a segment, which is obtained from the given one by extending its both ends by $\frac{\delta}{2}$ (see the constraint C3), if the given segment is of non-zero length. Such a relation allows for point-shaped objects to slide freely among other diagram objects, while path endpoints remain enclosed between the corresponding node sides.

The obstacle relation defines the *obstacle graph* of the segments. The obstacle graph is planar; therefore its edge number is small. Moreover, if extended segments have no common points, the edges of the obstacle graph may be directed from left to right, and it becomes a planar dag defining the basic ordering of layout segments.

The obstacle graph does not represent the complete ordering information. Some additional efforts have to be made to ensure the correct ordering for newly inserted nested nodes that are represented by single points. To guarantee constraint C2, a special procedure is called to separate overlapping path segments to avoid unnecessary path crossings resulting from inappropriate segment ordering. Overlapping path segments can be the result of the routing algorithm following the fastest routing strategy: for each rectilinear path to be routed other paths are not taken into account.

After a complete segment ordering is determined, it is also represented by a graph. Besides ordering information, we include arcs into this graph from the left segment of every node to its right segment. That ensures minimum node size constraints C1. We call the graph obtained the *constraint graph*. Like the obstacle graph, the constraint graph is a directed acyclic graph and is also small.

We have found that layout optimality may be expressed via a quadratic optimization problem:

minimize $F(x_1, x_2, \dots x_n)$

subject to $x_j - x_i \geq d_{ij} \geq 0$,

where $x_1, x_2, \dots x_n$ are the x-coordinates of the segments, and the pairs (i, j) are the arcs of the constraint graph.

The function F is built to minimize the changes of the layout, and in its most usual form is a sum comprising summands of two kinds and corresponding only to diagram nodes.

To minimize the node drift, we introduce the summands

$$(\frac{x_l + x_r}{2} - x_c)^2,$$

where for each node x_l, x_r are the abscissas of its left and right segments, and x_c is a constant abscissa of its old center.

To minimize the node size, F comprises also summands of the form

$$w \cdot (x_r - x_l)^2.$$

The weighting factor w should be chosen in an appropriate way. The value 10 seems good enough.

After the minimization problem has been solved, the diagram is recalculated for the new places of the segments (Fig. 3b).

Analogously, the layout is processed in the vertical direction (Figs. 3b, 3c).

Because of real-time conditions, we need a fast algorithm for our optimization problem. It is shown in the next sections that in practice it may be solved in $\sim n^p$ time where $1.5 < p < 2$.

Optimization technique

As described above, we must deal with functions in the form

$$F(x) = \sum_k L_k^2(x), \quad (1)$$

where $L_k(x)$ denotes some linear function depending on an n-dimensional point $x = (x_1, x_2, \dots x_n)^T$. We need to minimize F subject to the inequality

$$Ax \geq d, \quad (2)$$

where each row r of the $m \times n$ matrix A comprises only two non-zero elements –1 and +1 in columns i_r and j_r respectively, and all the pairs (i_r, j_r) form an acyclic graph.

We have chosen the gradient projection method [M89] as the theoretical background for solving this quadratic programming problem. In its general form the method involves matrix computations in the case of linear constraints. We completely avoid matrix processing by exploiting the simplicity of our constraints.

The solution is found in two stages. At first a feasible starting point x_0 satisfying the inequality $Ax_0 \geq d$ is searched. If such a point exists, then our problem obviously has a solution.

Lemma 1. *The set of feasible points is non-empty*.

Proof. Let us number the vertices of the constraint graph topologically, and let d_{max} be the maximum component of the m-tuple d. Setting $x_i = i \cdot d_{max}$ we obtain x satisfying the condition (2). □

In fact, the topological sorting procedure may be slightly modified in order to transform an arbitrary point x into a feasible one much better than obtained by the proof of Lemma 1.

After the starting point is found, iterations are performed in order to find the solution. At each iteration the current point x is changed so that F decreases.

We have to distinguish two major cases: the inequality (2) is strong or not.

Case $Ax > d$.

In this case the point x is strongly inside the feasible area and we may shift x in the direction of the steepest descent $g = (-\nabla F(x))^T$.

We find two scalar values:

τ_1 minimizing the function $f(\tau) = F(x + g\cdot\tau)$, $\tau \geq 0$, and

$\tau_2 = \max (\tau \geq 0 \mid A\cdot(x + g\cdot\tau) \geq d)$.

Finding both τ_1 and τ_2 is easy because since (1) $f(\tau)$ is a quadratic function, and (2) is reduced to m linear inequalities of one variable.

Then x has to be changed to $x + g\cdot\min(\tau_1, \tau_2)$.

Case $Ax \geq d$, and equality holds for at least one dimension.

In this case the point x is on the border of the feasible area and we must shift x along the border in the direction which is the projection p of g onto the border.

To calculate p let us introduce a new $m_0 \times n$ matrix A_0 as the submatrix of A consisting of those rows of A for which strong equalities in (2) take place. Let d_0 be the corresponding subcolumn of d. We call the corresponding subgraph of the constraint graph the *active constraint graph* and denote it by G_0.

Lemma 2. *All vertices of every connected subgraph of G_0 have mutually equal corresponding projection components.*

Proof. From the choice of A_0 we have $A_0x = d_0$, and for an arbitrary shift y along the border defined by A_0 we have $A_0\cdot(x + y) = d_0$, too. Hence

$$A_0y = 0, \tag{3}$$

and consequently $A_0p = 0$.

The last equality means that for an arbitrary row of A_0 we have $p_i = p_j$, i.e. all arcs of G_0 have equal projection components for both ends. The required statement follows immediately. □

Lemma 3. *Let S be the index set of vertices of an arbitrary maximum connected subgraph of G_0, and, as stated above, all its vertices have the same projection component p_S. Then*

$$p_S = \frac{1}{|S|}\sum_{k\in S} g_k \ .$$

Proof. As p is the projection of g, $g - p$ is perpendicular to all directions y along the border. Because of (3), $g - p$ can be expressed as some linear combination of rows of A_0, i.e. taking an appropriate m_0-tuple u

$$g - p = A_0^T u. \tag{4}$$

The k-th row in the last equality is $g_k - p_k = \sum_{i=1}^{m_0} a_{ik}u_i$, where a_{ik} denotes an element of A_0.

We have $\sum_{k\in S}(g_k - p_k) = \sum_{k\in S}\sum_{i=1}^{m_0} a_{ik}u_i = \sum_{i=1}^{m_0} u_i \sum_{k\in S} a_{ik}$, and, since S includes either none or both ends of G_0's arcs, $\sum_{k\in S} a_{ik} = 0$ because each row of A_0 comprises exactly two non-zero elements –1 and +1.

Hence $\sum_{k\in S}(g_k - p_k) = 0$, and $\sum_{k\in S} g_k = \sum_{k\in S} p_k = |S|\cdot p_S$. □

Lemmas 2 and 3 allow us to calculate p from g in a very simple way. At first, we divide all components of g into subsets corresponding to the maximum connected subgraphs of G_0. Secondly, we calculate the average of the corresponding components of g.

After p is calculated, we have to distinguish another two cases.

Case $p \neq 0$.

In this case like in the case $Ax > d$ we find two scalar values:
τ_1 minimizing the function $f(\tau) = F(x + p\cdot\tau)$, $\tau \geq 0$, and
$\tau_2 = \max(\tau \geq 0 \mid A\cdot(x + p\cdot\tau) \geq d)$.

And then change x to $x + p\cdot\min(\tau_1, \tau_2)$.

Case $p = 0$.

This is the case when we have to change the matrix A_0 or stop the iterations. Because of the convexity of our optimization problem, the Kuhn-Tucker conditions allow us to distinguish between two cases.

From (4), we have

$$g = A_0^{\mathrm{T}} u. \tag{5}$$

The Kuhn-Tucker conditions mean that if there exists u satisfying (5) and

$$u_i \leq 0,\ i = 1, 2, \dots m_0, \tag{6}$$

then the optimum is reached.

Lemma 4. *Let all vertices of G_0 be partitioned into two disjoint subsets V and $\overline{V}$ in such a way that all arcs joining V and $\overline{V}$ go from V to $\overline{V}$ thus forming a directed cut separating V and $\overline{V}$. Let $\overline{S}$ be the index set of vertices of $\overline{V}$. If the cut is* positive, *i.e.* $\sum_{k\in\overline{S}} g_k > 0$, *then every u satisfying* (5) *violates* (6). *Besides, g is directed inside the feasible area relatively to its border defined by those rows of A_0, which correspond to the arcs of the cut, and the projection of g onto the feasible area's border defined by the other rows of A_0 is not equal to* 0.

Proof. Let C be the index set of rows of A_0 corresponding to the arcs of the cut.

Since each row of A_0 comprises exactly two non zero elements –1 and +1 that indicate the endpoints of the arc corresponding to the row, and since only arcs of the cut have exactly one (marked with +1) endpoint belonging to $\overline{V}$, we have $\sum_{k\in\overline{S}} a_{ik} = \begin{cases} 1, & \text{if } i \in C \\ 0, & \text{otherwise} \end{cases}$.

Hence, if u satisfies (5), $\sum_{k\in\overline{S}} g_k = \sum_{k\in\overline{S}} \sum_{i=1}^{m_0} a_{ik} u_i = \sum_{i=1}^{m_0} u_i \sum_{k\in\overline{S}} a_{ik} = \sum_{i\in C} u_i$, and $\sum_{i\in C} u_i > 0$ because of the given inequality. Obviously, for some i $u_i > 0$, i.e. (6) does not hold.

To prove that g is directed inside the feasible area relatively to its border defined by those rows of A_0, which correspond to the arcs of the cut, we show that there exists u satisfying (5) such that $u_i \geq 0$ for all $i \in C$.

Assume first that G_0 is connected and our cut is a minimum cut i.e. any proper subset of its arcs does not form a cut. In such a case there exists a spanning tree in G_0 that includes exactly one arc from our cut. We remove from G_0 all arcs of the cut except the one of the spanning tree, and we remove from A_0 the corresponding rows, thus obtaining the graph G_0' and the matrix A_0'. Besides, let for an $(m_0 - |C| + 1)$-tuple u' $g = A_0'^{\mathrm{T}} u'$.

In the graph G_0' the vertex sets V and $\overline{V}$ are still separated by a positive directed cut. Hence, by the same argument as for u, the unique component of u' corresponding to the cut is positive.

It is easy to see that the required u is obtainable from u' by setting all missing components to 0.

In the case when G_0 is disconnected or our cut is not a minimum one, those parts of the cut, which are minimum cuts, must be examined separately in each maximum connected subgraph of G_0.

Finally, let us show that the projection of g onto the feasible area's border defined by those rows of A_0, which do not correspond to the arcs of our cut, is not equal to 0.

Denote by G_1 the graph obtained from G_0 after removing all arcs of the cut. Some of G_1's maximum connected subgraphs constitute the part $\overline{V}$. Let S_j ($j = 1, \dots$) be the vertex index sets of these subgraphs:

$\overline{S} = S_1 \bigcup \ldots$. As S_j are mutually disjoint and $\sum_{k \in \overline{S}} g_k > 0$, some of the sums $\sum_{k \in S_j} g_k$ must be different from 0. Because of Lemma 3, this means the required property. □

Lemma 5. *If for every directed cut separating V and* $\overline{V}$ *in* G_0 $\sum_{k \in S} g_k \leq 0$ *holds, then there exists u satisfying* (5) *and* (6).

Proof. Let us extend G_0 by adding two new vertices *s* and *t*, and by adding additional arcs from *s* to all vertices with $g_k \geq 0$, and from all vertices with $g_k < 0$ to *t*. We are about to pass a flow through the extended graph. The capacity of the original arcs is set to ∞, and the capacity of all arcs adjacent to *s* or *t* is the value $|g_k|$ corresponding to the second end of the arc.

There exists a flow with a value $g^+ = \sum_{g_k \geq 0} g_k$. To prove this, we have to verify the well-known Ford-Fulkerson condition: the total capacity c_{in} for all ingoing arcs of every set $V \bigcup \{t\}$ must be at least g^+, where *V* is subset of the vertices of G_0.

If at least one arc from G_0 goes into *V*, then $c_{in} = \infty > g^+$.

In the opposite case G_0 has a directed cut separating *V* and $\overline{V}$.

Let *S* and $\overline{S}$ be the index sets of vertices from *V* and $\overline{V}$ respectively, and

$$g_S^+ = \sum_{k \in S,\, g_k \geq 0} g_k \ , \ g_{\overline{S}}^- = \sum_{k \in \overline{S},\, g_k < 0} g_k \ .$$

It is clear that $g^+ = g_S^+ + g_{\overline{S}}^+$, and by the condition of the Lemma $g_{\overline{S}}^+ + g_{\overline{S}}^- \leq 0$.

Since only arcs going into $V \bigcup \{t\}$ are adjacent to *s* or *t*, $c_{\text{in}} = g_S^+ - g_{\overline{S}}^- = g^+ - g_{\overline{S}}^+ - g_{\overline{S}}^- \geq g^+$.

Thus a flow with a value g^+ exists and gives the values $\varphi_i \geq 0$, $i = 1, 2, \ldots m_0$ to arcs of G_0.

Let $In(k)$ and $Out(k)$ denote the index sets of ingoing and outgoing arcs of *k*th vertex of G_0. It holds $In(k) = \{i \mid a_{ik} > 0\}$, $Out(k) = \{i \mid a_{ik} < 0\}$.

It is easy to see that for our flow we have $g_k = \sum_{i \in In(k)} \varphi_i - \sum_{i \in Out(k)} \varphi_i = \sum_{i:\, a_{ik} < 0} \varphi_i - \sum_{i:\, a_{ik} > 0} \varphi_i = \sum_{i=1}^{m_0} a_{ik} \cdot (-\varphi_i)$.

Hence $g = A_0^{\mathrm{T}}(-\varphi)$, where $\varphi = (\varphi_1, \varphi_2, \ldots \varphi_{m_0})^{\mathrm{T}}$. □

Lemmas 4 and 5 show how to distinguish in the case $p = 0$ between changing the active constraint graph or stopping the iterations. If there exists a positive directed cut, iterations must be continued beforehand removing the rows corresponding to the arcs of the cut from A_0.

To test the existence of such a cut is the most complex part of our optimization method. Fortunately, the question is well-studied [H97] and can be solved by the maximum flow technique. The proof of Lemma 5 is just based on the corresponding construction.

The gradient projection method works well at our application. Nevertheless, it may be made significantly faster due to the very clear geometric background of the problem. Indeed, according to Lemmas 2 and 3, when we shift the current point *x* to its new position, each maximum connected component of the active constraint graph moves as a rigid body. We have observed that there is no need to move all components simultaneously by the vector $g \cdot \min(\tau_1, \tau_2)$. Any direction where *F* decreases is admissible. We can take the components one by one and shift them in a direction, which decreases the function. If two components touch each other we merge them. The outline of the algorithm follows:

(1) Shift and merge components of the active constraint graph while possible;
(2) Calculate a positive cut;
(3) If such a cut exists, remove its arcs from the active constraint graph and continue with (1).

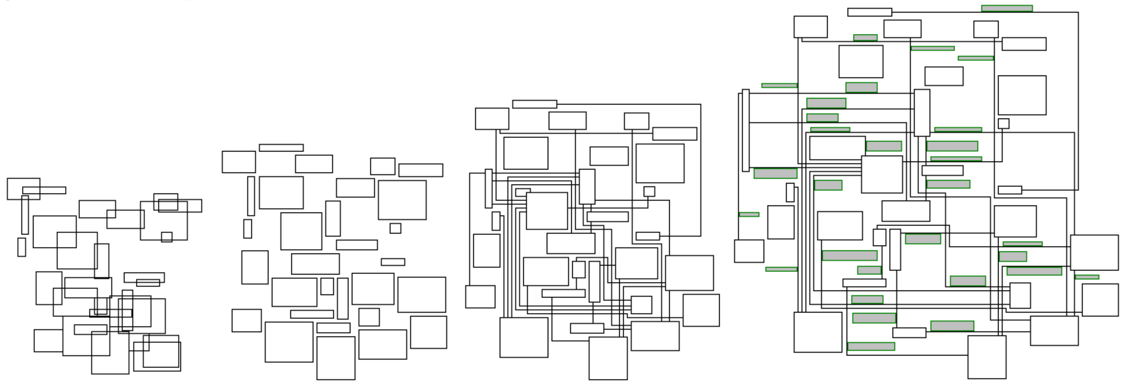
Furthermore, we can get rid of costly flow computations by maintaining a spanning tree in each component. At each merge we update the tree by adding one active arc between the two components. The necessary cut of the tree can be calculated in linear time.

After these modifications the algorithm converges significantly faster than the direct implementation of the gradient projection method based on Lemmas 1–5.

In the next section we give examples of the practical behavior of our approach.

Application examples

The main and most important application example is diagram normalization. To measure the time complexity of our optimization method, we generate series of realistic-looking diagram examples randomly in the following way. We take N random upright rectangles representing diagram nodes. Placing them randomly, they may intersect (Fig 4a). To obtain a correct diagram, intersections must be eliminated. This task is solved by our technique giving the node layout (Fig 4b). Next we add N random independently routed paths. Independent routing may generate path segments with violated minimum distance requirements, which are made correct by *Normalize* (Fig 4c). As the last step we add path labels, freeing the required space using once more *Normalize* (Fig 4d). In all steps the mental map coming from the initial rectangle positions is preserved.



a **b** **c** **d**

Figure 4. Initial rectangles (a), rectangles after intersection elimination (b), normalized random paths (c), and random size path labels (d).

Basically the preservation of the mental map is expressed as minimization of node drift subject to obstacle graph requirements. We can use different models as well, for example preserving orthogonal ordering as discussed in [MELS95, HIMF98]. In our technique we accomplish this by adding arcs between adjacent rectangles (with respect to the ordering) to our constraint graph. Fig. 5 shows the rectangle intersection elimination while preserving the orthogonal ordering applied to the same starting position (Fig 4a).

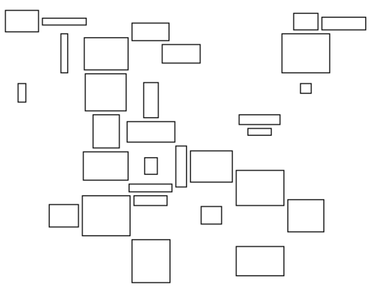


Figure 5. Rectangles of Fig 4a after intersection elimination while preserving the orthogonal ordering

Tables 1 and 2 show the performance of the C++ implementation of our optimization method running on a PENTIUM 120MHz computer. The average data from ten examples is taken. The first table reflects diagram processing illustrated in Fig. 4. The second one shows elimination of rectangle intersections on larger datasets in two cases: while preserving the orthogonal ordering, and with obstacle graph approach. The segment count (n), iteration count (I) and time in seconds (T) is given in x and y directions separately.

In all examples the generated rectangles are placed in a square at a density where the total area of the rectangles approximately equals the area of the square. When normalizing in horizontal direction, the rectangle height is taken two times smaller. To obtain a better solution of the two-dimensional problem it is possible to do normalization several times gradually increasing rectangle height when normalizing in horizontal direction. However, our experience shows that the quality improvements are not significant.

Table 1. Diagram processing.

Intersection elimination				
n	I_x	T_x	I_y	T_y
1000	9	0.1	18	0.3
2000	11	0.3	29	0.9
3000	16	0.6	38	1.6
4000	16	0.9	48	2.7

Path routing					
n_x	I_x	T_x	n_y	I_y	T_y
5610	37	3.1	5545	37	2.9
13140	60	12.7	13001	46	9.2
21616	66	21.7	21427	58	18.1
30899	82	38.3	30640	75	33.0

Labeling					
n_x	I_x	T_x	n_y	I_y	T_y
6610	10	1.3	6545	21	2.2
15140	9	2.7	15001	26	6.9
24616	10	4.4	24427	34	14.2
34899	11	7.3	34640	36	22.6

Table 2. Rectangle intersection elimination.

	Obstacle ordering				Orthogonal ordering			
n	I_x	T_x	I_y	T_y	I_x	T_x	I_y	T_y
1000	8	0.1	17	0.2	17	0.2	24	0.3
2000	12	0.3	26	0.7	27	0.7	37	1.0
4000	16	1.0	46	2.8	43	2.4	60	3.4
8000	24	3.0	80	10.6	74	8.8	105	13.0
16000	38	8.6	145	35.2	123	31.6	193	51.5

The performance obtained in our experiments can be expressed as $\sim n^p$, where $1.5 < p < 2$ depending on the problem type.

An interesting observation is that after a few iterations of the optimization, visual changes of the diagram are negligible; therefore we can stop the iterations. Indeed, our previous version [KR96] is essentially finding a feasible point without optimization, followed by post-processing to shrink unnecessarily expanded nodes. Although cutting off iterations gives considerable time cut, we do not use this since our diagrams usually contain not more than a few thousand segments, where the method is fast enough. In addition even in large diagrams small interactive changes done by the user require only a few iterations since the starting point is close to the optimum.

Conclusions

The optimum layout adjustment technique has been developed to handle graph-like diagrams of complex structure at the lowest level. Our *normalization* concept has turned out to be very powerful, allowing creation of a layout of a diagram in several stages. An independent path routing followed by normalization leads to a quite flexible system. We can use the same routing algorithms as those used in interactive editing. Further, the node layout stage does not have to consider the paths in great extent. We can process the most complex path structures including forks afterwards. The known algorithms do not deal with forks at all or demand some simplifying conditions. For example, [S97] requires forks to form an acyclic graph. We do not have such requirements because of handling forks as *supports*.

Many high-level operations are essentially based on our optimization technique, like layout *compaction* and *correction*. Correction is a *Normalize* like procedure that can get the constraints C1…C6 satisfied. We only have to replace all nodes by zero-sized rectangles and calculate a correct constraint graph. Compaction is another analogue of *Normalize*. It reduces distance between nodes by minimizing some other cost function. The degree of compaction can be easily controlled, even in the opposite direction thus expanding the layout.

There can be parts that require hierarchical structuring (forks, directed paths) and parts, which can be laid out unrestricted in our diagrams. When the node layout is generated automatically, diagrams are reduced to graphs and we combine two intermediate-level algorithms one for laying out undirected graphs and one for directed graphs.

An undirected graph we lay out on the grid, repeatedly moving each vertex to a free gridpoint closest to the barycenter of its neighbors. To ensure free gridpoints near the desired place, we expand the layout time after time, by compacting it and inserting empty rows and columns.

Since we use grid, we have to note that our optimization technique solves the corresponding integer programming problem with practically good approximation by simply rounding off the real-valued solution. More important is that we are not restricted to a quadratic function, a linear one is also allowed. Besides, in the integer linear case we get the exact result because of simplicity of our constraints.

A directed graph, possibly containing vertices representing supports and edges corresponding to fork paths, we lay out conventionally in a layered structure [GKNV93, DETT99]. If the graph contains cycles then the number of edges going upward is minimized and temporarily reversed, thus getting an acyclic graph. First vertices are placed into layers and then ordered inside these layers to minimize edge crossings. In both cases our optimization technique is involved in the following way.

In accordance with [GKNV93, DETT99], the optimum placement of vertices into layers minimizes the total vertical extent of all edges, i.e. the sum of differences between layer numbers of edge vertices. Taking our temporary acyclic graph as the constraint graph and requiring the vertical extent of each edge to be at least 1, we immediately obtain an integer linear programming problem, which is solved as mentioned above.

Further, vertices are ordered inside layers according to their neighbor barycenters. To keep the vertices separated, we just call *Normalize*. After the vertex order is determined, we assign the final horizontal coordinates by minimizing the total edge length squares as suggested in [DETT99]. Of course we sum up only squares of the horizontal extents of the edges, and minimize this sum subject to the constraints coming from an already found extremely simple vertex ordering. Despite the caution given in [DETT99], our optimization technique does not require considerable computational resources and solves the problem efficiently.

Another and particularly important stage of the layout creation is path label assignment. Maintaining textual labels on the paths is a hard problem managed only by few systems [KR96, DKMT98, G]. Our approach essentially facilitates the situation by partitioning it into two independent and technically simpler subproblems: looking for free places, and deforming the layout if there is not enough place. Having good initial label positions, *Normalize* provides their correct size preserving the initial mental map, thus obtaining quite pretty path labeling [G].

Acknowledgements

The authors would like to thank the GRADE development group at the Institute of Mathematics and Computer Science of the University of Latvia for enabling them to participate in an interesting collaborative project, and particularly Kārlis Podnieks and Andris Zariņš.

The work was supported by the Latvian Council of Science under grant 96.0247 and by ESPRIT project No 23287 ADDE.

References

[BNT86] C. Batini, E. Nardelli, R. Tamassia. A layout algorithm for data flow diagrams, – *IEEE Trans. Software Eng.*, vol. SE-12, no. 4, 1986, pp. 538–546.

[BRJ99] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide, – Addison-Wesley, 1999.

[BT98] S. Bridgeman, R. Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms, – Proc. of Graph Drawing ’98, *Lecture Notes in Computer Science*, vol. 1547, 1998, pp. 57–71.

[BDPP99] G. Di Battista, W. Didimo, M. Patrignani, M. Pizzonia. Orthogonal and quasi-upward drawings with vertices of prescribed size, – Proc. of Graph Drawing '99, *Lecture Notes in Computer Science*, vol. 1731, 1999, pp. 297–310.

[DETT94] G. Di Battista, P. Eades, R. Tamassia, I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography, – *Computational Geometry: Theory and Applications*, vol. 4, no. 5, 1994, pp. 235–282.

[DETT99] G. Di Battista, P. Eades, R. Tamassia, I. G. Tollis. Graph Drawing, Prentice Hall, 1999.

[DM90] C. Ding, P. Mateti. A framework for the automated drawing of data structure diagrams, – *IEEE Trans. Software Eng.*, vol. 16, no. 5, 1990, pp. 543–557.

[DKMT98] U. Dogrusoz, K. G. Kakoulis, B. Madden, I. G. Tollis. Edge labeling in the Graph Layout Toolkit, – Proc. of Graph Drawing '98, *Lecture Notes in Computer Science*, vol. 1547, 1998, pp. 356–363.

[GKNV93] E. R. Gansner, E. Koutsofios, S. C. North, K-P. Vo. A Technique for Drawing Directed Graphs, – *TSE* vol. 19, no. 3, 1993, pp. 214–230.

[G] Graphical Diagramming Engine, – *http://www.gradetools.com/*.

[HIMF98] K. Hayashi, M. Inoue, T. Masuzawa, H. Fujiwara. A layout adjustment problem for disjoint rectangles preserving orthogonal order, – Proc. of Graph Drawing '98, *Lecture Notes in Computer Science*, vol. 1547, 1998, pp. 183–197.

[HM97] W. He, K. Marriott. Constrained graph layout, – Proc. of Graph Drawing '96, *Lecture Notes in Computer Science*, vol. 1190, 1997, pp. 217–232.

[H97] D. S. Hochbaum. A new-old algorithm for minimum cut and maximum flow in closure graphs, – Technical Report, University of California, Berkeley, 1997.

[KR96] P. Ķikusts, P. Ručevskis. Layout algorithms of graph-like diagrams of GRADE Windows graphic editors, – Proc. of Graph Drawing '95, *Lecture Notes in Computer Science*, vol. 1027, 1996, pp. 361–364.

[KM99] G. W. Klau, P. Mutzel. Combining graph labeling and compaction, – Proc. of Graph Drawing '99, *Lecture Notes in Computer Science*, vol. 1731, 1999, pp. 27–37.

[LE95] T. Lin, P. Eades. Integration of declarative and algorithmic approaches for layout creation, – Proc. of Graph Drawing '94, *Lecture Notes in Computer Science*, vol. 894, 1995, pp. 376–387.

[MM88] J. Martin, C. McClure. Structured Techniques: The Basis for Case, – Prentice Hall, 1988.

[M89] M. Minoux. Programmation Mathematique, Theorie et Algorithmes Dunod, – Bordas et C.N.F.T. – E.N.S.T., 1989.

[MHT93] K. Miriyala, S. W. Hornick, R. Tamassia. An incremental approach to aesthetic graph layout, – *Proc. of Int. Workshop on Computer-Aided Software Engineering (CASE '93)*, 1993, pp. 297–308.

[MELS95] K. Misue, P. Eades, W. Lai, K. Sugiyama. Layout adjustment and the mental map, – *Journal of Visual Languages and Computing*, vol. 6, 1995, pp. 183–210.

[PSTS91] L. B. Protsko, P. G. Sorenson, J. P. Tremblay, D. A. Schaefer. Towards the automatic generation of software diagrams, – *IEEE Trans. Software Eng.*, vol. 17, no. 1, 1991, pp. 10–21.

[RDMMST87] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, A. Tuan. A browser for directed graphs, – *Software – Practice and Experience*, vol. 17, no. 1, 1987, pp. 61–76.

[SBKP98] U. Sarkans, J. Barzdins, A. Kalnins, K. Podnieks. Towards a metamodel-based universal graphical editor, – *Proc. of the Third International Baltic Workshop on Databases and Information Systems*, Riga, 1998, pp. 187-197.

[S97] J. Seemann. Extending the Sugiyama algorithm for drawing UML class diagrams: towards automatic layout of object-oriented software diagrams, – Proc. of Graph Drawing '97, *Lecture Notes in Computer Science*, vol. 1353, 1997, pp. 415–424.

[SM81] K. Sugiyama, K. Misue. Methods for visual understanding of hierarchical system structures, – *IEEE Trans. Syst. Man, Cybern.*, vol. SMC-11, no. 2, 1981, pp. 109–125.

[SM91] K. Sugiyama, K. Misue. Visualization of structural information: automatic drawing of compound digraphs, – *IEEE Trans. Syst. Man, Cybern.*, vol. 21, no. 4, 1991, pp. 876–892.

[TDBT88] R. Tamassia, G. Di Battista, C. Batini, A. Tuan. Automatic graph drawing and readability of diagrams, – *IEEE Trans. Syst. Man, Cybern.*, vol. 18, no. 1, 1988, pp. 61–78.