Visual Computer

An algorithm for blob hierarchy layout

David Harel,
Gregory Yashchin

Dept. of Applied Mathematics and Computer Science, The Weizmann Institute of Science, 76100 Rehovot, Israel
E-mail: harel@wisdom.weizmann.ac.il, greg.yashin@ness.com

Published online: 15 March 2002
© Springer-Verlag 2002

We present an algorithm for the aesthetic drawing of basic hierarchical blob structures, of the kind found in higraphs and statecharts and in other diagrams in which hierarchy is depicted as topological inclusion. Our work could also be useful in Web page design, window system dynamics, and possibly also newspaper layout, etc. Several criteria for aesthetics are formulated, and we discuss their motivation, our methods of implementation and the algorithm's performance.

Key words: Aesthetics – Hierarchy – Higraph – Layout – Statechart

1 Introduction

1.1 Background and motivation

Graphics and diagrams play an important role in representing data in various fields of science, engineering and visual interfaces. Since graphic elements are usually intended to be comprehended by humans, readability is often the prime target of a designer. Achieving a clear, aesthetic picture of a given diagram is not an easy task when carried out manually, so layout algorithms are badly needed in these areas.
Several graphic standards or conventions have been proposed for representing data of different nature. "Classical" graphs are usually composed of nodes, represented by dots or boxes, and edges, represented by straight, polygonal or curved lines, which can be either directed or non-directed. Trees play a special role due to their ability to symbolize hierarchical structures. Some less "classical" graphical notations are Euler circles, Venn diagrams, hypergraphs and higraphs.
A particular "non-classic" notation we are interested in is that of *higraphs* [Harel 1988]. Higraphs consist of *blobs* (rounded-corner rectilinear shapes – but we shall concentrate on rectangles here), possibly connected by edges. The blobs are arranged in an inclusion hierarchy. In a general higraph, blobs may intersect, and blobs of different hierarchy levels may be connected. Different versions of higraphs are suitable in a variety of cases, including sets and relations, inclusion hierarchies, and specification languages such as statecharts [Harel 1987], used in the design of complex reactive systems. Higraphs with non-intersecting blobs and without edges are also known in the literature as inclusion graphs or treemaps [Shneiderman 1990].
Our work focuses on developing and implementing drawing algorithms for blob hierarchies, which are edge-free and intersection-free higraphs, with an attempt to produce the most aesthetic, and therefore comprehensible, layouts.
The need for this arises in many applications involving extensive interaction between computer and human by means of diagrammatic languages. For example, the user of STATEMATE [Harel et al. 1990, Harel, Politi 1998], RHAPSODY [Harel 1988], or some of the UML diagrams [OMG 1999] faces the task of actually drawing statecharts, activity charts and object diagrams. All these are actually interpreted variants of higraphs. Similar issues arise in other software development tools that support encapsulated hierarchies. Automation of statechart layout,

for example (at least in some limited way), would help the user concentrate on the design and save time spent on technicalities; it could also help the viewer understand the system structure that is being described. A good solution to the hierarchy layout problem can be a starting point for such improvements, but it would definitely need extending to the case where edges are also present, as in the languages supported by such tools.

Another possible application of a solution to this problem is in window managers of graphical user interfaces and in the design of Web pages. Many existing interfaces of commercial applications supply options of automatic arrangement of windows in a tile or in a cascade, as do many kinds of Web pages. Usually, after the operation all the windows are of the same size. None of these interfaces cares for window contents or inner structure, and they ignore any inclusion hierarchy. Integration of ideas from our algorithms in such applications could make it easier to improve the layout of window structure, displaying the windows simultaneously and in appealing dimensions and locations. Other possible applications are in newspaper layout and similar design tasks.

1.2 Related work

Motivation for the higraph formalism can be found in [Harel 1988]. Several authors present applications of them: [Harel 1987] starts a series of papers on statecharts (see also [Harel, Gery 1997, Harel et al. 1990, Harel, Politi 1998]), which are used for the specification and design of reactive systems, and are also a central part of the UML standard [OMG 1999]. Higraphs can be used to visualize hierarchical systems, which have attracted quite a bit of attention. A higraph-based description of file system security constraints was proposed in [Maimone et al. 1990]. Blob-like structures are used in [Shneiderman 1990] for the description of hierarchical information structures, such as directory trees [Turo, Johnson 1992] or NBA basketball player statistics [Turo 1994].

Despite all this, consulting the comprehensive survey [Di Battista et al. 1999], one notices that there are almost no attempts to develop algorithms for drawing higraph-like structures, although there is strong motivation for visualizing hierarchical structures [Eades et al. 1993a, Eades et al. 1993b, Eades et al. 1996, Eades, Feng 1996, Eades et al. 1998, Kant et al. 1996, Metaxas et al. 1994]. Existing research on higraph-like drawing addresses only limited versions of the problem. The algorithms presented in [Eades et al. 1993a, Eades et al. 1993b] work on inclusion relations defined by binary/ternary trees (each blob has at most two or three sub-blobs inside). The algorithms presented in [Eades et al. 1998] concentrate on planarity and edge layout problems in clustered graphs. The space filling approach described in [Shneiderman 1990, Turo 1994] appears to be suitable only for describing resource allocation hierarchical distributions, where the most important visualized feature is the weight associated with each blob. The problem of automatic layout of a windows hierarchy is hardly addressed in existing window managers. However, extensive work is being done to improve the user's manual control over the layout (see, for example, [Kandogan, Shneiderman 1996]).

2 Criteria for aesthetics

The objective is to arrange an inclusion hierarchy of non-overlapping blobs in a planar layout constituting an aesthetic representation that achieves the best positioning of the involved blobs. The only constraint here is to preserve the structure hierarchy. We are allowed to move, resize and rearrange the blobs in any possible way, as long as they stay rectangular and no borders are ever crossed. To give a feeling for the structures we are working with and the desired outputs, Fig. 1 contains the results of our algorithm run on a nontrivial hierarchy containing seven generations of blobs.

This short and simple formulation of the problem hides a great difficulty, the one behind the word "best". Thus, our goal is highly subjective.

We now discuss some general criteria for the quality of a solution.

The criteria surveyed here resulted from considering how a human would approach the problem and what the basic constraints guiding him or her would be. However, the final and exact definitions for each criterion were arrived at after numerous attempts to encode and implement the criteria, and after applying the algorithm to many problem instances of varying complexity. Still, in many cases we found ourselves having to make hard choices of layout selection, feeling that a "right" choice depends on the viewer's personal taste and application requirements. Therefore,

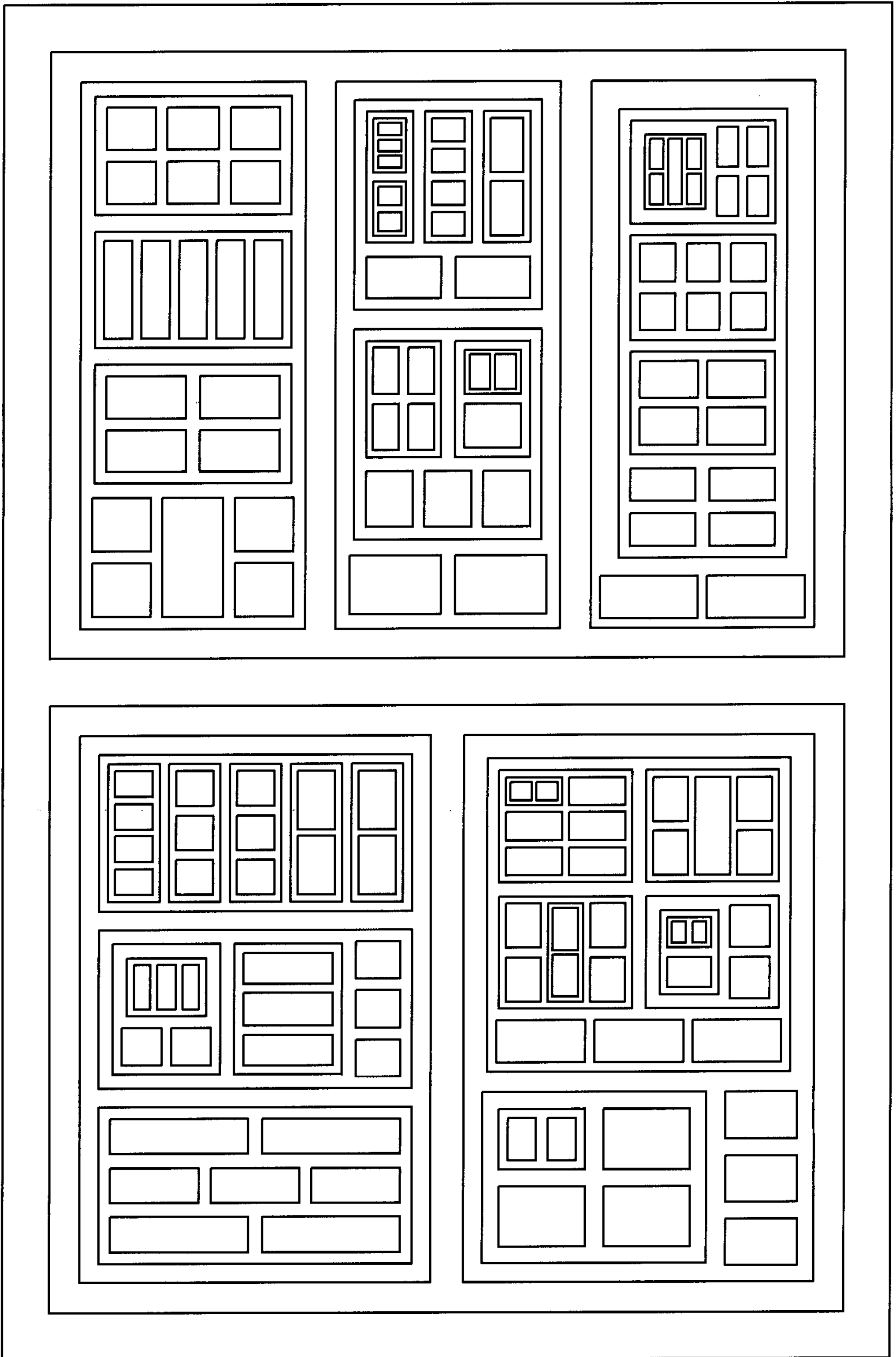


Fig. 1. The algorithm's result (with default parameters) on a depth-7 hierarchy

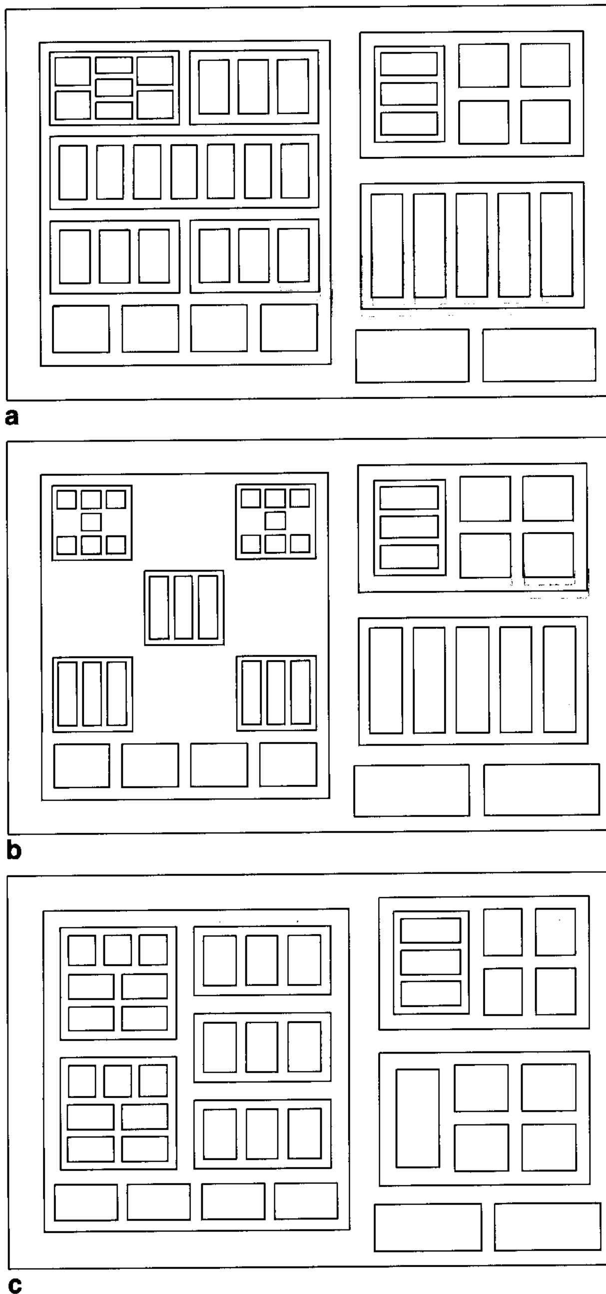


Fig. 2. A blob hierarchy layed out using three variants of our algorithm

in our implementation we have made many of algorithm's parameters user-adjustable, with a set of default values that, in our subjective opinion, yield the best results. Figure 2 presents an example of several layouts produced by three versions of our algorithm run on the same hierarchy. These versions approach the problem of space utilization in different ways, and each might be found suitable under particular circumstances (see Sect. 2.6)

We illustrate the importance of the various criteria, and describe the ways to formulate them algorithmically. Most of the examples presented in this section are actual screen shots of our algorithm's implementation.

One of the most intuitive ways to describe the input hierarchy is by a tree describing the inclusion relation: for example, Fig. 3 shows the tree corresponding to the hierarchies in Fig. 4. In this article we shall freely use the common notions of *root*, *leaves*, *parents*, *offspring*, *siblings*, *generations*, *ancestors* and *descendants*.

2.1 Uniformity of blob dimensions

A rather intuitive requirement from an aesthetic layout is for blobs of *similar "importance"* to have *similar size*. Exact formulation of this criterion requires the definition of these two terms:

Importance similarity

We define collections of siblings with *similar inner complexity* as those of similar importance. Every blob is assigned a *weight*, reflecting its inner structure. The weight is evaluated by counting blobs' descendants, which is done recursively, all the way down to the leaves. Siblings of the same parent are then divided into uniform weight groups and are treated as blobs of similar importance. Sections 3.2.1 and 3.2.2 describe the process in detail.

Another blob collection, orthogonal to those described above, is that of all the leaves. It was found that imposing uniformity of leaf size contributes to the clarity of the layout.

Perfect satisfaction of the two requirements of size uniformity is usually hard to achieve simultaneously, and we had to search for a reasonable compromise between them. Figure 4 displays the improvement achieved laying out a hierarchy according to these criteria (layout b), opposed to a more simple approach of defining siblings to always have similar importance (layout a).

Size similarity

Whenever possible, we try to achieve perfect size similarity by choosing identical dimensions. Usually

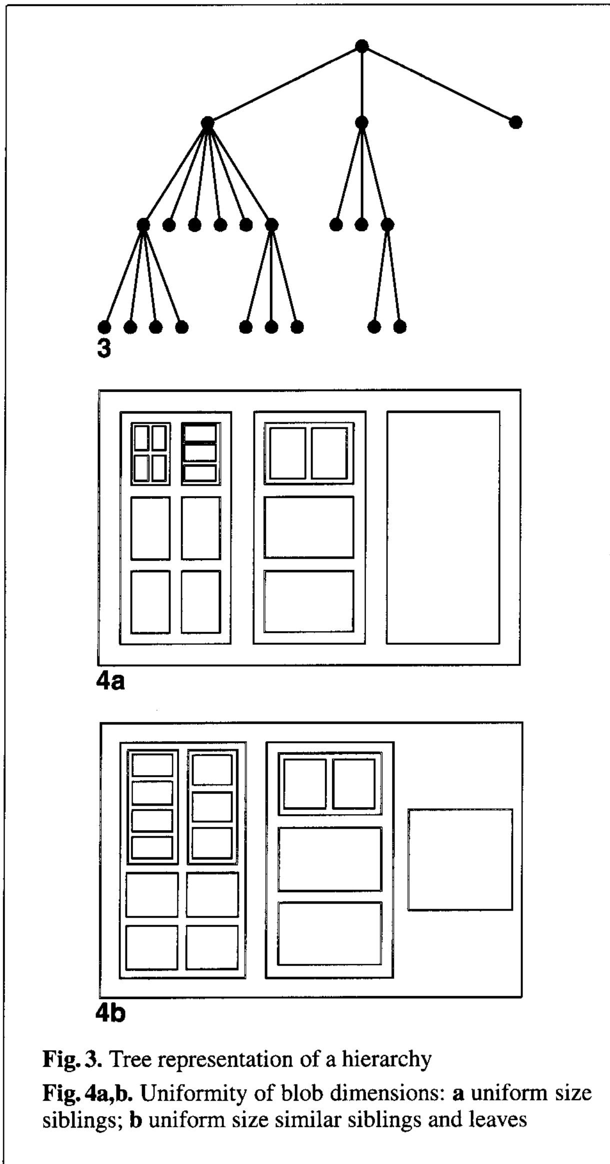


Fig. 3. Tree representation of a hierarchy

Fig. 4a,b. Uniformity of blob dimensions: **a** uniform size siblings; **b** uniform size similar siblings and leaves

this is easy to do among siblings, but for the entire set of all the leaves, we can only try to converge to something close to the best. This is why size similarity of a non-identical group of blobs has to be measured subject to some optimization.

Since the resource allocated by our algorithm is area, the simplest formulation of size similarity is to require uniformity of the areas of blob collections of similar importance. This can be done, for example, by measuring the standard deviation of area values in a given set. However, we found this to be unsatisfactory, since shape has a strong influence on our judgment of area. It is usually hard to accept a high and narrow rectangle to be of the same size as a square, even when their areas are equal.

Instead, we distinguish the group of *longer* edges (i.e., the sides of the rectangles) and the group of *shorter* ones and try to achieve uniformity in each of these. The smaller the standard deviation of each group is, the better our criterion is satisfied.

As already mentioned, size similarity of similarly important siblings is one of the basic features of our algorithm (excluding the special case of maximal space utilization, described in Sect. 3.3.1, where we compromise this criterion). Size similarity of leaves is maximized at a later stage (see Sect. 3.4).

2.2 Connecting inner structure and size

The connection between the inner structure of blobs and their size complements the discussion of blob dimensions of the previous section. Since we wish to give similar size to similar blobs, we would also like to use size to emphasize differences between blobs.

It sounds reasonable that blobs with more complicated inner structure should be larger. A blob's inner structure complexity is evaluated by counting the descendants, producing a *weight.*

We use the most intuitive recursive algorithm, which usually works fine. However, some caution must be taken when translating weight to size. We have found that the connection between a blob's weight and size should not be too sensitive. Small changes in a blob's importance (weight) should rather produce *identical* results and not just *similar* ones. Obtaining large identical size groups of blobs adds flexibility to the algorithm and helps satisfy the symmetry criteria discussed in Sect. 2.3. In order to achieve this we modify simple weighting by the following: First, the contribution of lower generations to the ancestor's weight is decreased by a constant factor. Second, the computed weight is quantized, so that blobs of *near* weights will actually receive the *same* size. The layout of Fig. 5b looks more organized and clear than that of Fig. 5a.

Utilization of the formulated principles in the algorithm is described in detail in Sects. 3.2.1, 3.2.2 and 3.2.3.

2.3 Symmetry

Symmetry is possibly the most logical criterion for aesthetics in an average human's opinion. One type

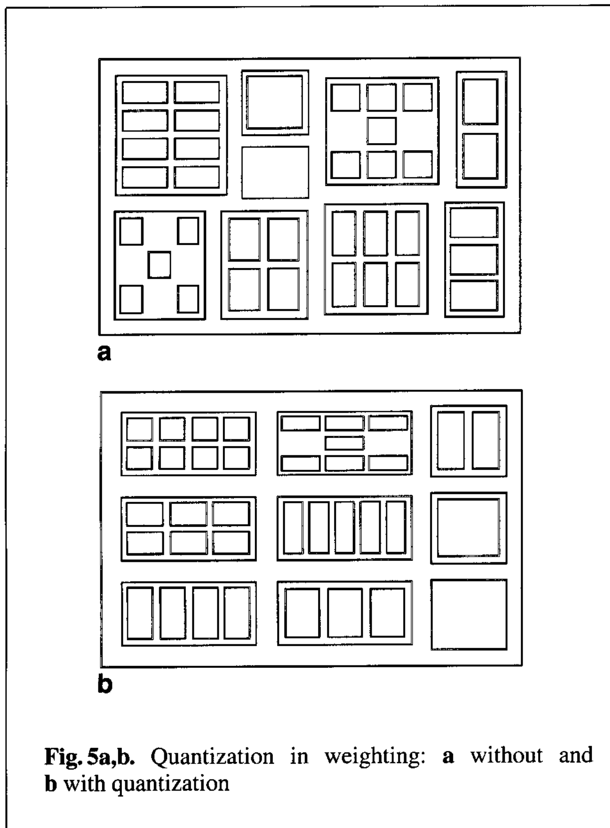


Fig. 5a,b. Quantization in weighting: **a** without and **b** with quantization

of symmetry that we would like to require of a good layout is derived from the nature of the geometric figures with which we deal. The layout, or at least its portions, should be linear symmetric about both axes, exactly like the rectangular blobs of which it consists.

Since the entire layout may be composed of blobs of different dimensions, it makes sense to request symmetry only in homogeneous portions of the entire hierarchy. These are groups of identically sized siblings, present in the layout in order to satisfy size uniformity and the inner structure criteria of Sects. 2.1 and 2.2.

2.4 Blob proportions

Humans have well-defined preferences for specific proportions of rectangles, i.e., the ratio between the edges. One agreed-upon "ideal" proportion is the Golden Ratio $[(1+\sqrt{5})/2 \approx 1.618]$. Other popular shapes are the ratio -$\sqrt{2}$ (which is the proportion of the A paper format), and the square. We have chosen the Golden Ratio as the ideal, but the choice could be made to depend on personal taste and constraints of the host application (such as screen ratio or printing page size).

Obviously, we would like to measure the quality of blob proportion according to how far it is from the ideal value. The comparison is carried out on a logarithmic scale rather than a linear one in order to fix asymmetry between ratios larger than 1 (vertical shapes) and those smaller than 1 (horizontal shapes): $||\log(\mathtt{X}/\mathtt{Y})| - \log(\mathtt{IDEAL_PROP})|$ measures the imperfectness of the shape. Here IDEAL_PROP denotes the ideal blob proportion, which is a parameter of the algorithm, satisfying $\mathtt{IDEAL_PROP} \geq 1$.

This criterion was used in the algorithm, with a slight change. We found that square-like shapes seem to be more pleasing to the eye than long and narrow or short and wide ones. More equally proportioned shapes are also preferable for most applications. Therefore, we have used the above criterion with different constant factors for disproportion of the two kinds, giving priority to square- like shapes.

2.5 Gap uniformity

It seems reasonable to set the gaps between identical siblings located in proximity to be uniform. As we observed, it is also important for gaps in *both directions* to be equal too, although the blobs are not square-shaped. Rather surprisingly, this also helps to enforce equality, or at least similarity, of gaps between blobs in *different generations*.

The layout of Fig. 6b is a more successful version of that of Fig. 6a. It was produced by imposing equality of gaps in both axes and trying to make gaps in different generations close, rather than naïvely choosing them to be proportional to the parent blob's size.

Satisfying the formulated criterion of gap equality in different generations is maximized in the optimization stage of the algorithm (see Sect. 3.4). The amount of space to allocate for gaps and additional details of the criterion satisfaction are described in Sect. 3.2.6.

We have also found several special cases where gap equality should be treated with care. Consider, for example, the hierarchy presented in Fig. 7. A "lighter" offspring of the hierarchy tree root seems to look much better if freed from the equal gaps criterion. Notice that in this case the gap uniformity principle contradicts leaf size uniformity (Sect. 2.1),

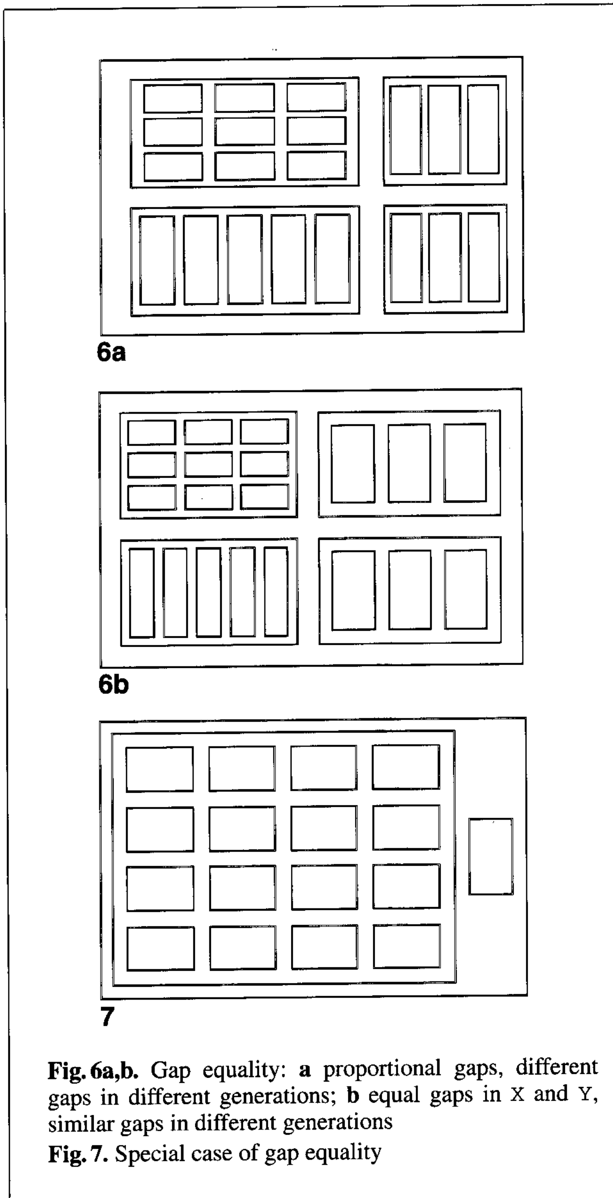


Fig. 6a,b. Gap equality: **a** proportional gaps, different gaps in different generations; **b** equal gaps in X and Y, similar gaps in different generations

Fig. 7. Special case of gap equality

and relaxing the gap uniformity contradicts the space utilization (Sect. 2.6) criterion.

2.6 Space utilization

Space utilization is the last aesthetic criterion we consider. It often contradicts other criteria we have stated, and one is sometimes tempted to compromise it. On the other hand, for most applications, the drawing area is an expensive resource, which we would like to utilize efficiently.

Contradiction with blob proportions

A simple hierarchy consisting of a root and 26 identical offspring leaves is presented in Fig. 8. Positioning the blobs on a 13×2 grid (layout a) utilizes the parent area optimally; however, blob proportion is unsatisfactory. Using a 6×5 grid (layout b) yields the best proportion, but more than 10% of the parent area is empty. A 9×3 grid (layout c) gives a reasonable compromise.

Contradiction with dimension uniformity

An alternative to the compromise of Fig. 8c is to ease the blob dimension uniformity constraint. If we allow similarly important siblings to have similar dimensions, and not necessarily identical ones, we might reach the solution presented in Fig. 8d. The space is fully utilized, and the element proportions are close to ideal. However, similarity of the siblings is not emphasized as strongly as by the layouts of Fig. 8a–c.

The layout of Fig. 8e presents another approach to solving the problem. Size identity is compromised even more, but the viewer has a stronger impression of the identity of sibling importance.

Contradiction with gap uniformity

The existence of gaps is a contradiction to area utilization. Clearly, some applications, such as window managers, would prefer to work with zero gaps, and use every available pixel of the screen. However, other applications would prefer to give up some of the valuable area for gaps to improve clarity of the layout. In Sect. 2.5, we discussed the importance of a good choice of gaps.

Conformance to symmetry

Unlike the previous three criteria, symmetry is better suited to space utilization. A natural way to obtain a symmetric, uniform and well-packed layout is to use an ordered structure fitting a rectangular container. This is why the central idea of our algorithm is to divide blobs into uniform groups and to use *rectangular equally spaced grids* as the basis for positioning identical offspring inside an area allocated for them (see Sect. 3).

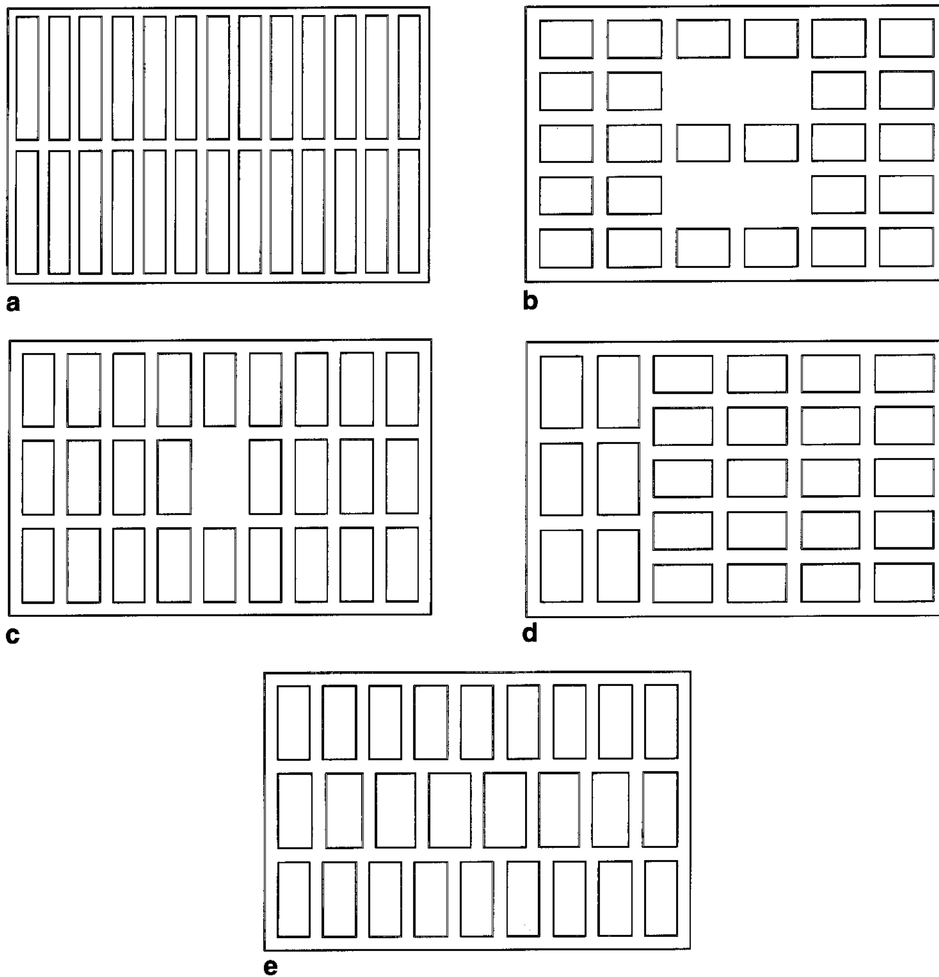


Fig. 8a–e. Contradiction between space utilization and blob proportions: **a** good utilization, bad proportions; **b** bad utilization, good proportions; **c–e** compromises

The algorithm presented in Sect. 3.2 implements this principle and tries to give the best compromise for the discussed contradictions. Section 3.3 presents two improvements to the basic algorithm, trying to fulfill the space utilization perfectly.

3 The algorithm

3.1 General structure of the algorithm

The algorithm works in two stages. The aim of the first stage is to produce a reasonable layout, which will be the basis for consequent processing. We execute several recursive procedures, starting from the root and proceeding down to the leaves in a depth-first manner. Due to the up-down non-iterative nature of this stage, there is not much cooperation between different branches of the tree. This is why it is rather difficult to satisfy the aesthetic criteria that require constraints to be global to the entire hierarchy, such as leaf size uniformity. Therefore, the produced layout mostly satisfies local requirements only. It supplies input to the next stage, optimization.

In this stage, the obtained layout is used as a starting point for a series of optimization rounds. We converge to a solution that satisfies both local and global

aesthetics criteria by varying several optimization parameters that affect the entire layout. No significant topological changes are allowed at this stage, so that perfect criteria satisfaction is rarely obtained. The blobs may change size and move a little, but no more complicated rearrangement is allowed. However, even so, most layouts yield significant aesthetic improvement.

3.1.1 Reasonable positioning

We have decided to divide our description of the first stage of the algorithm into two sections. In Sect. 3.2 we present most of this stage, for the time being relaxing the space utilization criterion. Section 3.3 introduces two additional optional steps that try to satisfy the latter criterion as well.
The following are the major steps of the first stage of our algorithm:

- *Weighting*. Each blob is assigned a weight that reflects its inner structure ("importance"). The weight will help the subsequent steps of the algorithm to assign it an appropriate area.
- *Grouping*. Blobs belonging to the same parent are grouped according to their weights to produce several uniform sized groups.
- *Partitioning*. Each blob divides the space it was granted between the groups of its offspring. A rectangular area is allocated for each group of identically sized offspring.
- *Grid dimension choice*. For each area of identical sized siblings, the underlying grid dimension is chosen.
- *Positioning*. Given a group of uniform weight blobs and an area, the blobs are positioned in the area on a chosen grid in a symmetric fashion, leaving some grid slots empty.
- *Gap computation*. Before the blobs can be placed in their positions on the grid, one has to define the size of the empty gaps that will separate neighboring blobs.

Two additional steps, described in Sect. 3.3, give two ways to optimize space utilization:

- *Group sub-division*. This step is applied to an area if none of the grids considered by the grid dimension choice were found to be satisfactory. The blobs are divided into two sub-groups, despite the fact that they have identical size. The partitioning step is reapplied to the problematic area to produce two sub-areas. Each of the two areas continues executing the algorithm separately.
- *Size adjustment*. This step is optionally applied after gap computation. In order to cover the empty grid cells, we adjust the dimension of blobs occupying the neighbors of the empty cell.

3.1.2 Optimization

This second stage of the algorithm implements a classical anti-gradient walk in the space of the optimization parameters. The iteration is carried out with gradually decreasing steps, up to convergence to a local minimum, which we hope is close to the global one. In Sect. 3.4, we discuss the details: the energy function, the optimization parameters and the optimization process.

3.2 Basic positioning

The notion of *area* is essential for the description of our algorithm. We have already mentioned that area is a sub-division of an inner space of a blob. The formal definition is recursive:

- The "inside" of a blob is an area.
- Each area contains either two sub-areas or an arbitrary number of identically sized blobs placed on a uniform spaced grid. If the area is the "inside" of a blob, it may also be empty.

An example of an area tree for a three-level blob hierarchy is shown in Fig. 9.

3.2.1 Blob weighting

The layout of a given hierarchy starts with the recursive step of blob weighting (described here) and sibling grouping (described in Sect. 3.2.2) executed in depth-first order. After the procedure is finished, every blob is assigned a `Weight` that will guide the following steps for space allocation.
Upon accepting a request from the parent, a blob asks all its offspring to weigh themselves and group their successors recursively. After that the offspring are grouped, and their weights are adjusted by the grouping process (see Sect. 3.2.2). The obtained weights are summed, and the total is multiplied by a `CHILD_WEIGHT` factor. The blob's own weight of 1.0 is then added. The final result is:

$$\texttt{Weight} = 1.0 + \texttt{CHILD_WEIGHT} \times \sum_{\texttt{Child} \in \texttt{Children}} \texttt{Child}\cdot\texttt{Weight}.$$

The factor `CHILD_WEIGHT` is a parameter of our algorithm. It reflects the relative contribution of an

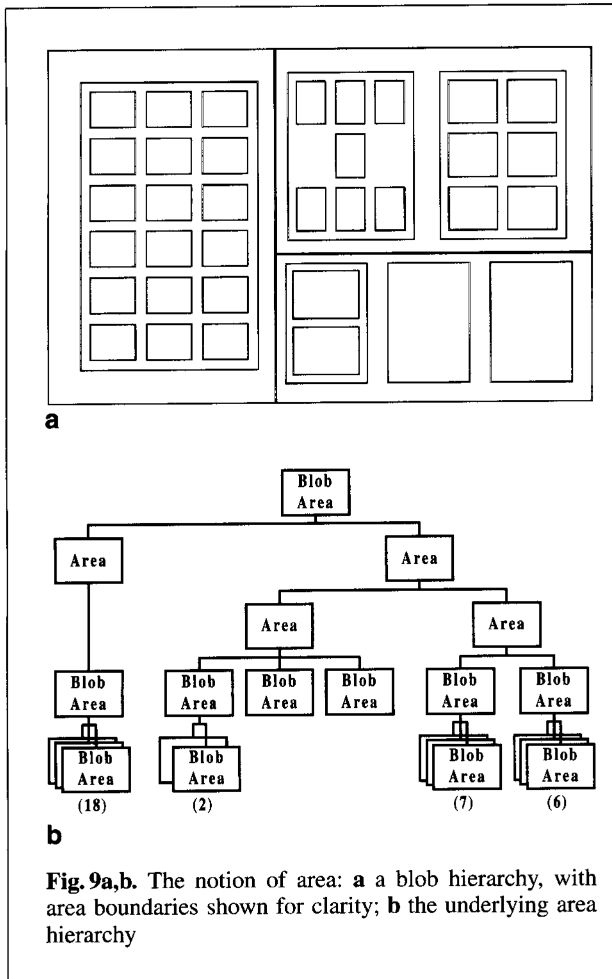


Fig. 9a,b. The notion of area: **a** a blob hierarchy, with area boundaries shown for clarity; **b** the underlying area hierarchy

offspring to the weight of its parent. Setting its value to 0.0 cancels the offspring's influence, so that all blobs belonging to the same generation get the same `Weight` = 1.0 (and therefore the same size – as in Fig. 4a). The value of 1.0 would make the algorithm extremely sensitive to the inner structure of a blob, which we found to be undesired (see Sect. 2.2). The default value of 0.5 that we have chosen was found to give good results for most hierarchies.

3.2.2 Sibling grouping

The grouping step takes place in the recursive weighting process right after the blob collects weight values from all its offspring. After the grouping is over, the offspring find themselves divided into identical weight groups. Later, each such group will get an area of its own (see Sect. 3.2.3) for its members to be placed inside on a grid (see Sects. 3.2.4 and 3.2.5). Since offspring weights might have numerous different values, we face the dilemma of either making small groups or correcting weights for the groups to be of a reasonable size. According to the discussion in Sect. 2.2, the latter option is preferable (see Fig. 5 for illustration).

The grouping process is carried out in the following fashion: First, the offspring are ordered by their weights. We then open the first group and start filling it up from the ordered list. We iterate through the list until we encounter an offspring whose weight differs from the weight of the first blob inserted into the group by the constant factor `GROUP_FACTOR` or more. Now, in order to achieve weight uniformity, all the weights of the group members are upgraded to the weight of the heaviest one. Then a new group is opened, and the process continues until all the offspring are processed. In Sect. 2.2 we referred to this process as *quantization*, which actually is logarithmic.

Consider, for example, grouping offspring with weights $\{10.0, 7.75, 6.25, 6.25, 3.75, 2.5, 2.25, 2.0, 1.25, 1.0, 1.0\}$. The algorithm will produce three groups of blobs: $\{4 \times 10.0\}$, $\{4 \times 3.75\}$ and $\{3 \times 1.25\}$, provided that `GROUP_FACTOR` = 2.0 (which is our default).

Choosing `GROUP_FACTOR` to be too small (with the extreme at 1.0) will cause the groups to include only identically weighted siblings, and to typically be small, producing results similar to those in Fig. 5a. Large values might cause the opposite result, in which groups are too large and contain blobs with very different inner complexity. The effect of this is similar to choosing `CHILD_WEIGHT` to be too low (see Sect. 3.2.1), with results such as those in Fig. 4a.

3.2.3 Area partitioning

After the weighting and grouping recursion is over, the root blob initiates another recursive procedure that results in a layout, i.e., each blob is eventually assigned its absolute coordinates. Each blob partitions its area among the offspring groups (described in this section), chooses grid dimensions for each area (Sect. 3.2.4), positions each group's members on the grid (Sect. 3.2.5) and computes the gaps between the blobs (Sect. 3.2.6).

The goal of the current step is to divide the blob's area into several rectangular areas, one for each

group. The size of each area will depend on the weight of the elements that are to be placed in it. This is done recursively, by repeatedly dividing an area into two sub-areas. If all of the area's offspring blobs belong to a single group, its entire area (its "inside") is simply granted to this group. This is the escape-clause of the recursion.

Given an area and a list of offspring organized into groups, the recursive step will be to divide the list into two sub-lists of weights as close as possible. Obviously, the division is carried out on the basis of whole groups only. After that, the area is split in two by a straight line parallel to an area edge.

The division of offspring is carried out by a simple greedy algorithm. We consider each group in the ordered list in turn and send it to one of the sub-lists, depending on which causes the two sub-lists to have closer weights.

After the blobs are divided into two sub-lists, the algorithm splits the area into two rectangular sub-areas: The total weights of the two sub-lists are computed. The longer edge of the parent area is then cut in two. We prefer to divide the longer edge of the area in order to stay with square-like shapes, which are more aesthetic (see Sect. 2.4) and give more flexibility in choosing grid dimension (described in Sect. 3.2.4). The process is somewhat reminiscent of the Slice- and-Dice algorithm for treemaps that appears in [Turo, Johnson 1992].

The division is a function of the ratio of the weights of the two sub-lists. In this way, the geometric areas of the two sub-areas will be controlled by their weights. Denote by `Edge` the longer edge of the parent area, by `Edge_1` the first half of it that will belong to the first sub-area, and by `Weight_1` and `Weight_2` the weights of the two sub-lists, respectively. Then the cut position is at

$$\texttt{Edge_1} = \texttt{Edge} \times \frac{\texttt{Weight_1} \times (1.0 - \texttt{MIN_AREAS_RATIO}) + \texttt{Weight_2} \times \texttt{MIN_AREAS_RATIO}}{\texttt{Weight_1} + \texttt{Weight_2}}.$$

Weighting by `MIN_AREAS_RATIO` limits the cut boundaries to be between

$$\texttt{MIN_AREAS_RATIO} \times \texttt{Edge} \quad \text{and}$$
$$(1.0 - \texttt{MIN_AREAS_RATIO}) \times \texttt{Edge}.$$

Here `MIN_AREAS_RATIO` is a parameter of the algorithm with $0.0 \leq \texttt{MIN_AREAS_RATIO} \leq 1.0$. Its default value is 0.1. This leaves both halves with reasonable size and proportion, even for extreme weight ratios.

3.2.4 Grid dimension choice

As mentioned in Sect. 2.6, each area positions its blobs at the nodes of a rectangular grid, which is evenly spaced. The following step of the algorithm determines the grid dimensions: `X_Dim` and `Y_Dim`. We also denote by `Offspring_Num` the number of blobs inside the area.

Clearly, we should only consider dimensions that satisfy $\texttt{Offspring_Num} \leq \texttt{X_Dim} \times \texttt{Y_Dim}$. Achieving equality would be ideal for optimal space utilization. However, `Offspring_Num` is not guaranteed to factor conveniently, or might even be a prime, which is why we also consider dimensions leading to inequality.

The algorithm loops through all the possible grids and evaluates their quality according to the aesthetics criteria. Due to reasons discussed in Sect. 3.2.5, two modes of work were defined for this step. In the `ARBITRARY` mode all the values of the dimensions are allowed, and in the `ODD_ONLY` mode even dimensions are forbidden and are corrected to the closest odd value.

The loop iterates through all the values of `X_Dim` and `Y_Dim` between 1 and $\lceil \sqrt{\texttt{Offspring_Num}} \rceil$. At each step,

$$\texttt{X_Dim} \times \left\lceil \frac{\texttt{Offspring_Num}}{\texttt{X_Dim}} \right\rceil \quad \text{and}$$

$$\left\lceil \frac{\texttt{Offspring_Num}}{\texttt{Y_Dim}} \right\rceil \times \texttt{Y_Dim}$$

grids are considered. Figure 10 displays some of the grids checked for an area containing 26 blobs.

For each grid, the algorithm computes a `Penalty`, reflecting the degree of violation of the blob proportion and space utilization criteria:

$$\texttt{Penalty} = \texttt{PROP_WEIGHT} \times \texttt{Prop_Penalty} + \texttt{SPACE_WEIGHT} \times \texttt{Space_Waste}.$$

Here, the proportion deficiency is evaluated according to Sect. 2.4 as follows:

$$\texttt{Prop_Penalty} = \left| \left| \log\left(\frac{\texttt{X_Edge/X_Dim}}{\texttt{Y_Edge/Y_Dim}}\right) \right| - \log(\texttt{IDEAL_PROP}) \right|.$$

In order to give priority to square-like shapes, this value is multiplied by the `DISPROP_WEIGHT` pa-

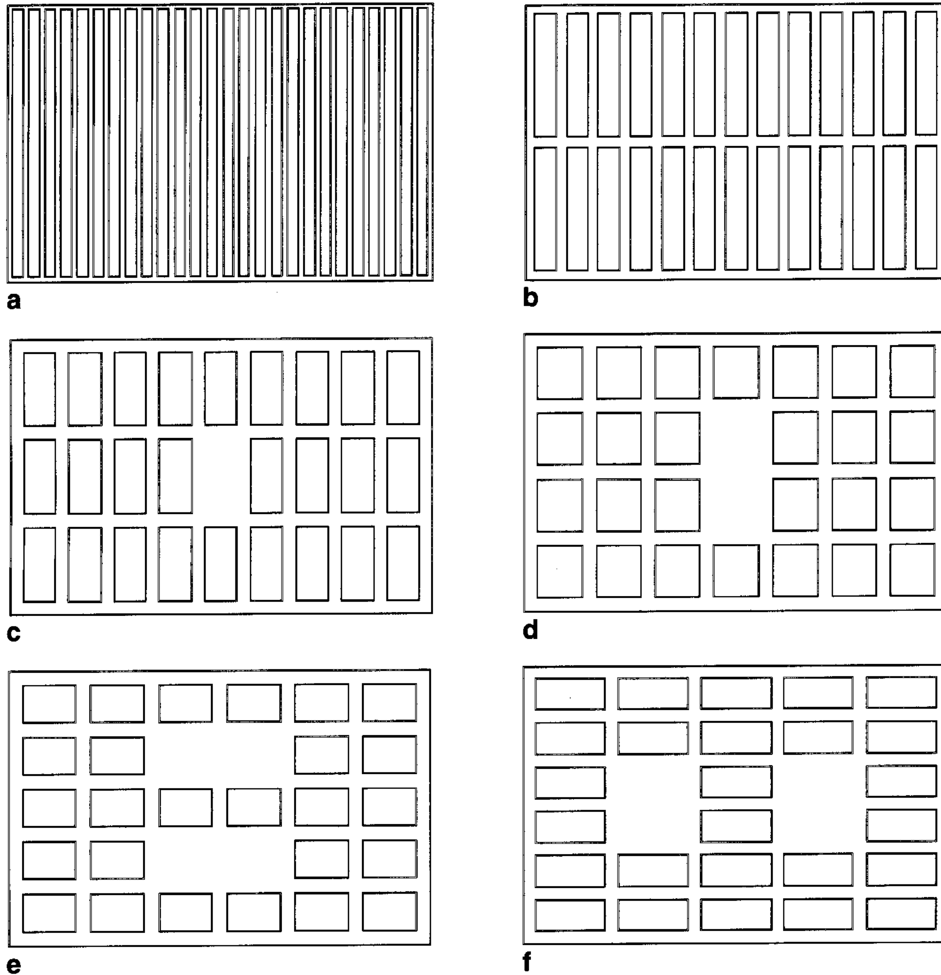


Fig. 10a–f. Possibilities in grid dimension choice: **a** 1×26; **b** 2×13; **c** 3×9; **d** 4×7; **e** 5×6; **f** 6×5

rameter if

$$\left|\log\left(\frac{\texttt{X_Edge/X_Dim}}{\texttt{Y_Edge/Y_Dim}}\right)\right| > \log(\texttt{IDEAL_PROP}).$$

The space waste is computed as

$$\texttt{Space_Waste} = 1 - \frac{\texttt{Children_Num}}{\texttt{X_Dim} \times \texttt{Y_Dim}}.$$

Both `Penalty` components are designed to be normalized and independent of the blob size. Weight defaults are $\texttt{DISPROP_WEIGHT} = 2.5$, $\texttt{PROP_WEIGHT} = 1.0$, and $\texttt{SPACE_WEIGHT} = 3.0$.

A grid with the minimal penalty is chosen and is then passed on to further steps of the algorithm. For the example of Fig. 10, the algorithm chooses layout c as the best. Out of the two `Penalty` components, we found the space utilization to be the more important one, so it was given a heavier weight. This is why layout c was preferred over layouts e and f, even though the latter have better blob proportions.

3.2.5 Positioning on the grid

The positioning algorithm step is responsible for obtaining a symmetric structure over the grid chosen in the previous section, similar to those shown

in Fig. 10. Given `Offspring_Num` and a grid `X_Dim` × `Y_Dim`, the algorithm chooses the grid positions that will stay unoccupied by the blobs.
As stated in Sect. 2.3, the basic requirement is symmetry about both axes. Unfortunately, this constraint might be impossible to satisfy in certain cases. Consider, for example, odd `Offspring_Num` and even grid dimensions. In a symmetric layout, one of the blobs must be positioned in the center of the grid, but in this case there is no such available grid place.

Claim: *If we restrict ourselves to grids of odd dimensions only, it is always possible to layout any number of blobs symmetrically (given that the grid is large enough).*

Proof (sketch): The proof is by induction.
In the case of `Offspring_Num` = 0 or 1, the blobs are trivially placed in a symmetric fashion on any odd dimension grid by either leaving it empty or allocating the central place.
At the inductive step we fill either the outer columns or rows, or the entire grid border (if there are enough blobs). Thus, the problem is reduced to placing a smaller number of blobs on a smaller inner part of grid still having odd dimensions. ■

The positioning algorithm works recursively according to the inductive framework of the proof. Given `N`, `X` and `Y`, we choose one of the possibilities of reducing the problem to a smaller grid, and make a recursive call. Figure 11 demonstrates the steps as the algorithm works on 29 blobs placed on 5 × 7 grid.
In fact, it is possible to achieve a symmetric layout on even-dimension grids too (as in Fig. 10a,b,d–f), so such grids should not be totally disregarded. The described algorithm is capable of working with grids of arbitrary dimensions, but it might fail to reach the end of the recursion. Facing an odd number of blobs to position, we try odd-dimension grids only. An even number of blobs will cause the algorithm to check all the possibilities, but certain grids will be discarded after failing the positioning step.
Figure 12 demonstrates positioning outcomes for certain numbers of siblings that do not factor nicely.

3.2.6 Gap computation

In Sect. 2.5 we discussed the importance of a good choice of inter-blob gaps and mentioned the difficulty of making the right choice in a non-iterative manner. The current algorithm step takes care of the reasonable choice of gaps in a single area without taking into account the situation in the other areas. We rely on the optimization step to fine-tune the layout, although the results of the first basic step are usually satisfactory.
First, the gap computation step determines what part of the parent's dimensions is to be left for gaps. The immediate result of the gap uniformity criterion is to use the same predefined *absolute* size gap in all the areas. However, this is impossible to implement, since a dense- enough population of blobs in an area can easily use up all the available space for gaps only. The next natural attempt is to allocate total gap size to be *proportional* to the area's linear dimensions. Unfortunately, the results of doing so were not satisfactory: The gaps were too large in areas with small grid dimensions, and were hardly seen in dense areas. The best results were achieved using the following *function* for producing the relative space left for the gaps (`Gaps_Percent`) depending on the number of gaps (`Gaps_Number`):

$$\texttt{Gaps_Percent} = \texttt{ASYMP_GAP} - \frac{1}{\frac{1}{\texttt{ASYMP_GAP}-\texttt{INIT_GAP}} + (\texttt{Gaps_Number} - 2)}.$$

This function returns values between the two constant parameters: `INIT_GAP` (for `Gaps_Number`= 2, a single row or column) and `ASYMP_GAP` (`Gaps_Number` → ∞). The layouts in Fig. 12 demonstrate the effect of the function, using default values of `INIT_GAP` = 0.15 and `ASYMP_GAP` = 0.3. The absolute gap size is slightly larger in areas with a small number of offspring, reflecting the relatively large size of offspring themselves. The gaps become smaller as the number of blobs grows, but enough space is still left between neighbors even in dense populations (see Fig. 12i with 140 blobs).
The space allocated for gaps is distributed uniformly across the grid. Special care is taken to deal properly with the "side gaps", i.e., the ones between the area boundaries and the exterior columns and rows. In the area of a full blob, these gaps should be as large as the gaps between the neighboring offspring. However, in an area that is an inner division of another area, this gap should only be half the size. The right-hand area of the root blob in Fig. 13 (the one containing 10 blobs) is of this kind.
The final correction to be made at this stage is to make gaps in both directions equal. Whenever possi-

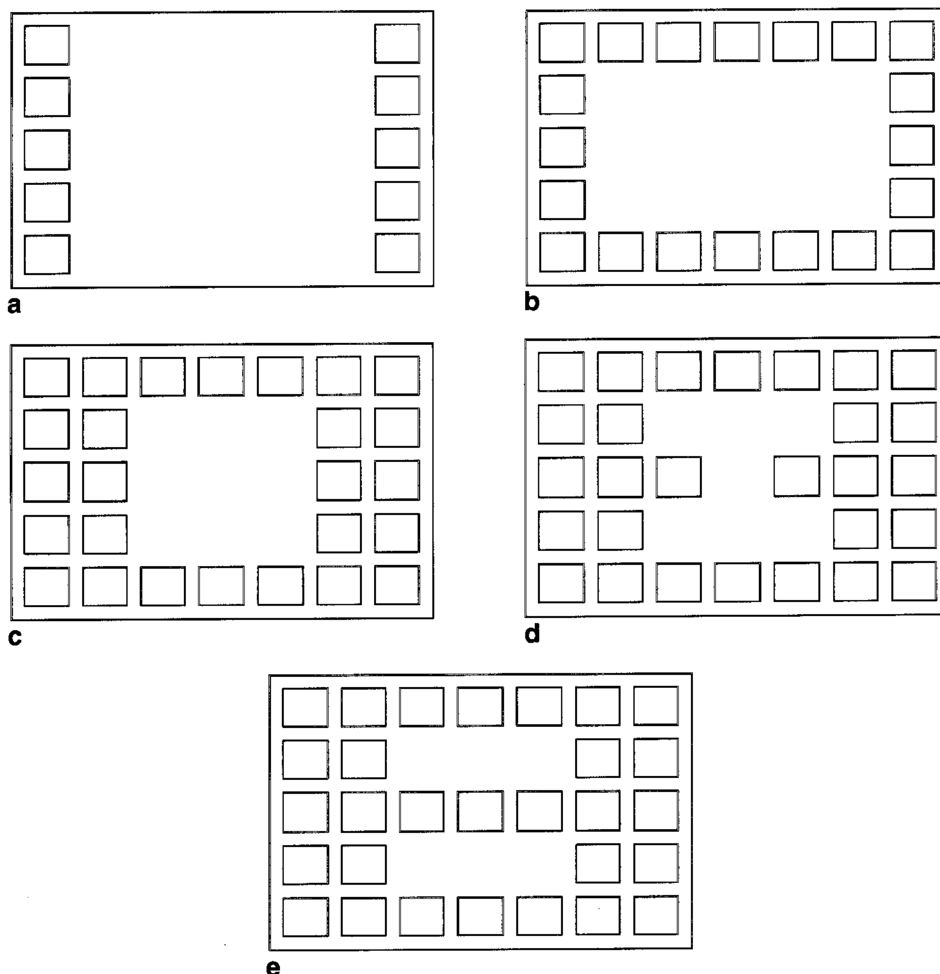


Fig. 11a–e. Positioning 29 blobs on a 5×7 grid: **a–e** the five steps of the algorithm

ble, we cause the gaps to be equal to the larger one, with the following exceptions:

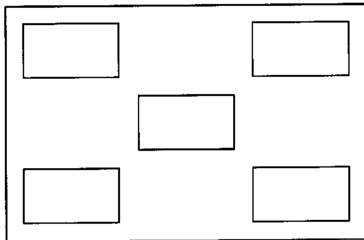
- If the larger gap is too large for the other dimension, then by adopting it we might exceed the maximum: `Gaps_Percent` > `ASYMP_GAP`. In this case the common-size gap is taken to be `Edge` × `ASYMP_GAP/Gaps_Number`.
- As mentioned in Sect. 2.5, the case of a leaf, which is a single offspring in the parent area, should also be handled with care. Gaps are not made equal if the parent area is a sub-area (such as in Fig. 14a), but if it is a blob's "inside", they are made equal (Fig. 14b).

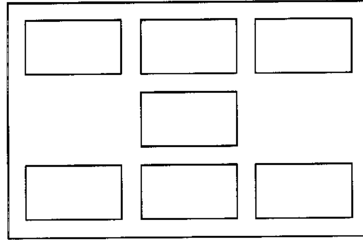
3.3 Maximal space utilization

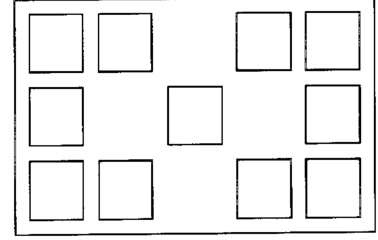
This section describes two additions to the first stage of the algorithm that were designed to improve utilization of the space in the layout. The general sequence of weighting, division into uniform groups and placement on a rectangular grid remains without change, but now no grid point is allowed to stay unoccupied.

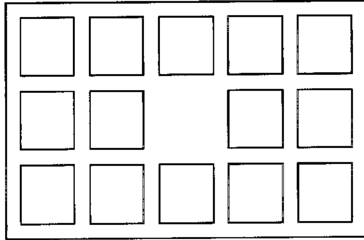
3.3.1 Group sub-division

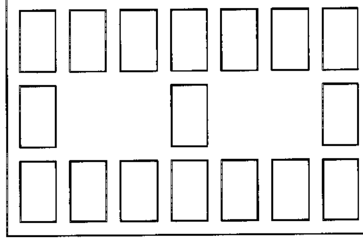
The group sub-division solution changes the grid dimension choice step (see Sect. 3.2.4) and dis-

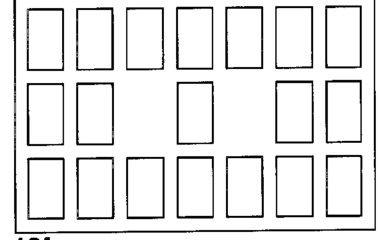

12a

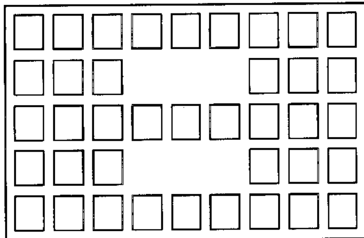

12b

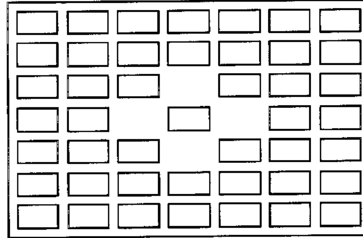

12c

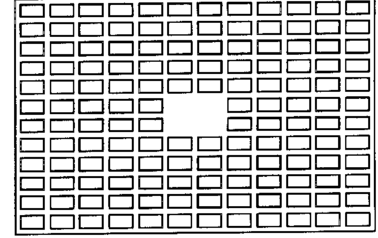

12d

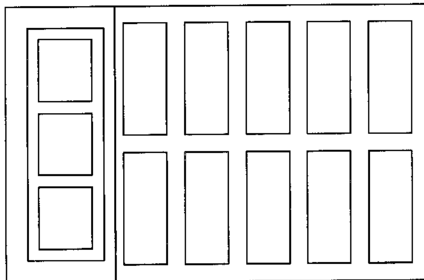

12e

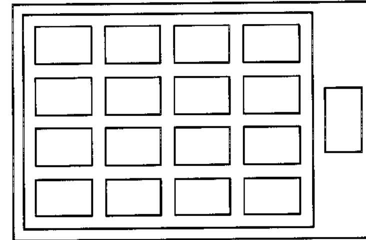

12f


12g


12h


12i


13


14a

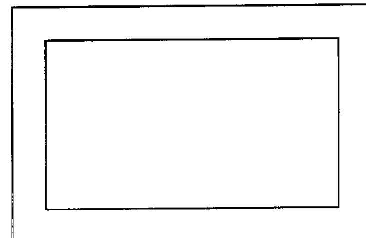

14b

Fig. 12a–i. The positioning stage for various numbers of blobs: **a** 5; **b** 7; **c** 11; **d** 14; **e** 17; **f** 19; **g** 39; **h** 45; **i** 140

Fig. 13. Side gap size demonstration

Fig. 14a,b. Gaps around a single offspring in an area: **a** different gaps case; **b** equal gaps case

cards any grid that cannot be filled perfectly, i.e., if $\texttt{Offspring_Num} < \texttt{X_Dim} \times \texttt{Y_Dim}$. Certainly, this narrows the space of possible solutions, and the best solution might be unsatisfactory.

The chosen solution `Penalty` is checked against the `MAX_PENALTY` constant threshold. If the chosen grid fails the test, the algorithm returns to the area partitioning step (see Sect. 3.2.3), in an attempt to partition the area into two and to divide the offspring between the two sub-areas, trying to find better grids for them both.

Since now all the blobs to be partitioned are of the same size, we are free to choose the sub-group cardinalities arbitrarily. The algorithm checks all the possibilities of partition between

$$(1, \texttt{Offspring_Num} - 1) \quad \text{and} \quad \left(\left\lfloor \frac{\texttt{Children_Num}}{2} \right\rfloor, \left\lceil \frac{\texttt{Children_Num}}{2} \right\rceil\right).$$

For each division, we check blob dimensions in the two sub-areas (X_1/Y_1 and X_2/Y_2, respectively). We try to satisfy the blob dimension uniformity and blob proportion criteria that are compromised by this approach. The two blob proportions are compared to each other and to the `IDEAL_PROP`, by computing:

$$\left|\left|\log\left(\frac{X_1}{Y_1}\right)\right| - \log(\texttt{IDEAL_PROP})\right| + \left|\left|\log\left(\frac{X_2}{Y_2}\right)\right| - \log(\texttt{IDEAL_PROP})\right| + \left|\left|\log\left(\frac{X_1}{Y_1}\right)\right| - \left|\log\left(\frac{X_2}{Y_2}\right)\right|\right|.$$

The lowest value wins.

Figure 15 shows the sub-divisions considered by the algorithm in an attempt to lay out 11 blobs. The best configuration found was layout b. Notice that we disallow recursive sub-divisions; these could have improved, for example, layouts d and e. We found that such attempts usually only make things worse, by violating the uniformity criteria even more; they are also highly time- consuming.

3.3.2 Size adjustment

The size adjustment portion of the algorithm tries to address the same space utilization problem as group sub-division, by allowing blob resize. An additional step is added after blob positioning (see Sect. 3.2.5). Logically, the blobs stay in the grid nodes they were assigned to, but their boundaries are moved in order to cover all the unoccupied squares. Figure 16 demonstrates the improvement to several of the layouts of Fig. 12, which had space utilization problems.

The algorithm loops through all the empty places on the grid and chooses blobs from the same row or column to participate in filling the space. Actually, it is enough to check a quarter of the area only and then rely on symmetry. Blobs in the chosen group are "blown up" in the appropriate direction (in a column vertically, and in a row horizontally). We found that it is best to do this gradually, so that the row or column blobs that are further from the empty space grow less, and the closer ones grow more.

Figure 17 contrasts this approach with two other less successful attempts. Figure 17a blows up only the neighbors adjacent to the empty space. If we consider the blow-up amount to be a function of the blob's distance from the empty space, this approach yields a step function. In Fig. 17b, the entire row is enlarged uniformly, and the blow-up function is a constant. Figure 17c demonstrates the option we have preferred, where the blow-up function is linear and blow-up factors form an arithmetic progression.

The algorithm works in the following fashion: For each empty place, we count the number of continuously placed blobs in the same row or column in each direction, that have not yet been used for space fillup; call these numbers `Up_Exist`, `Down_Exist`, `Left_Exist` and `Right_Exist`, respectively. If there are unoccupied neighboring places, the length of the unoccupied area is counted too; call these `Up_Empty`, `Down_Empty`, `Left_Empty` and `Right_Empty`, respectively. The entire area will be covered in a single move. In an attempt to minimize the impact on the size of the neighbors, we check how much space is to be covered relative to the number of blobs that will contribute to the job. The following values are thus computed:

$$\frac{\texttt{Up_Exist} + \texttt{Down_Exist}}{\texttt{Up_Empty} + \texttt{Down_Empty}},$$

$$\frac{\texttt{Left_Exist} + \texttt{Right_Exist}}{\texttt{Left_Empty} + \texttt{Right_Empty}}.$$

The column is chosen if the first value was found to be smaller, and the row is chosen otherwise.

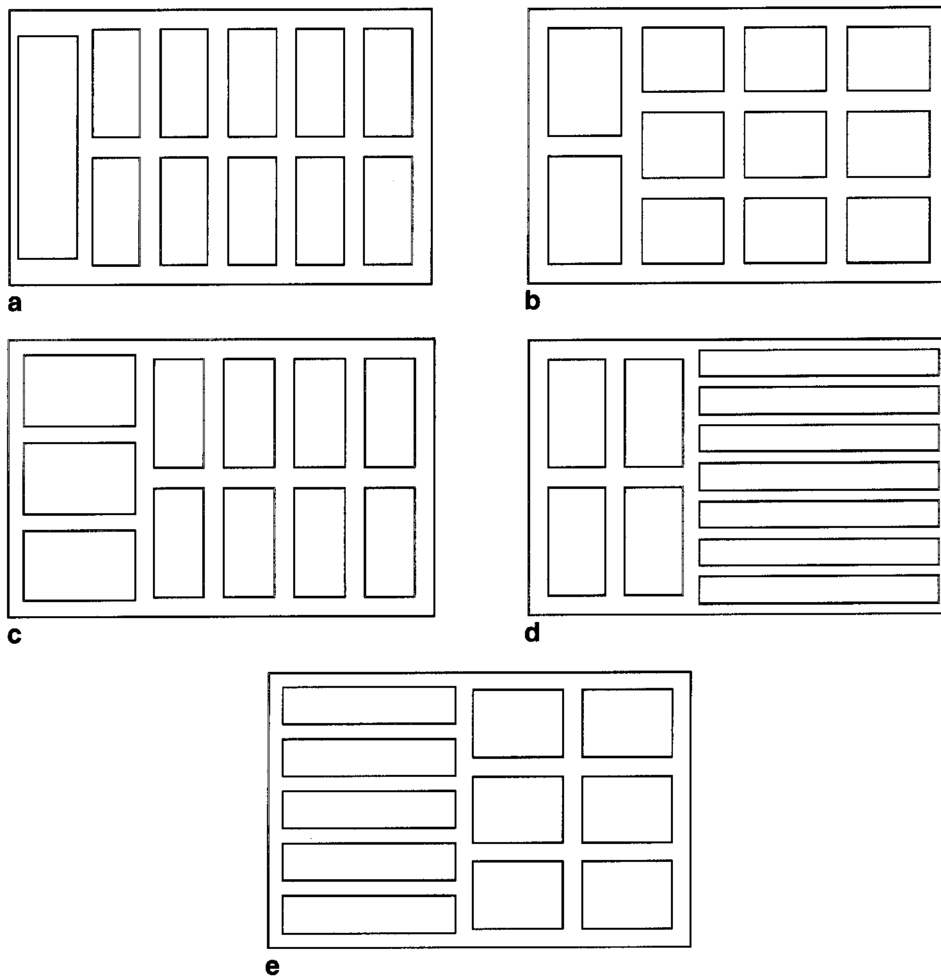


Fig. 15a–e. Group sub-division for 11 blobs: **a** 1 + 10; **b** 2 + 9; **c** 3 + 8; **d** 4 + 7; **e** 5 + 6

Now suppose, without loss of generality, that the horizontal direction (row) was chosen. The column case is symmetric. The total of `Left_Empty` + `Right_Empty` grid cells are divided between the left `Left_Exist` and the right `Right_Exist` neighbors in the row, proportionally:

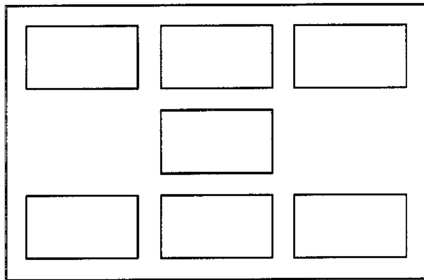
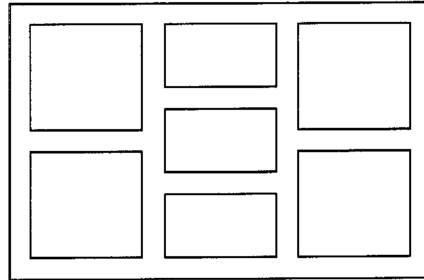
$$\texttt{Left_Fill} = (\texttt{Left_Empty} + \texttt{Right_Empty}) \times \frac{\texttt{Right_Exist}}{\texttt{Left_Exist} + \texttt{Right_Exist}},$$

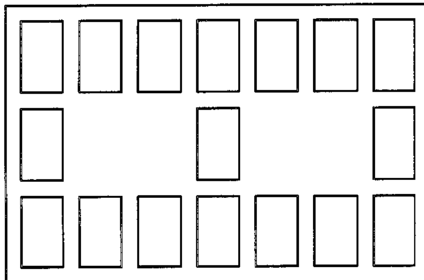
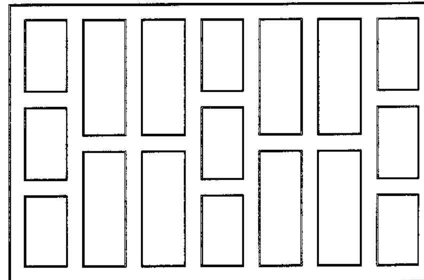
$$\texttt{Right_Fill} = (\texttt{Left_Empty} + \texttt{Right_Empty}) \times \frac{\texttt{Left_Exist}}{\texttt{Left_Exist} + \texttt{Right_Exist}}.$$

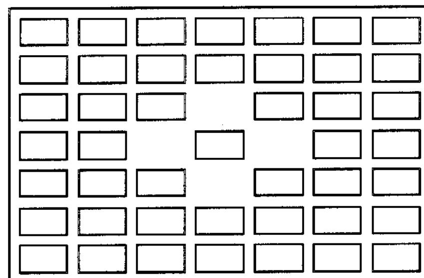
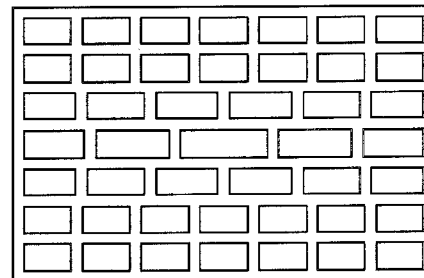
The following resizing sequence moves and enlarges the space covered by the left `Left_Exist` blobs by the total of `Left_Fill` grid cells. For each `I` in [1 ... `Left_Exist`], the `I`th left blob's left boundary moves to the right by (`I` − 1)/`Left_Total` of the grid step, and its right boundary by `I`/`Left_Total`. Here, `Left_Total` denotes

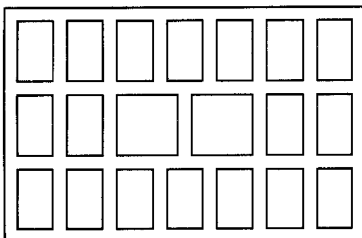
$$\frac{\texttt{Left_Exist} \times (\texttt{Left_Exist} + 1)}{\texttt{Left_Fill} \times 2},$$

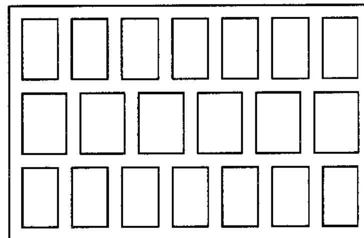
the sum of the arithmetic progression. Symmetrically, `Right_Fill` grid cells are filled up by the right half of the row.



16a



16b



16c


17a


17b

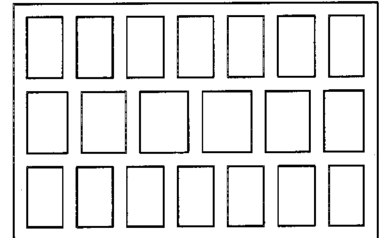

17c

Fig. 16a–c. Size adjustment: **a** 7 before and after; **b** 17 before and after; **c** 45 before and after
Fig. 17a–c. Size adjustment approaches: **a** step function; **b** uniform function; **c** linear function

3.4 Optimization

The second main step of the hierarchy layout algorithm, optimization, receives the layout obtained from the first step (described in Sects. 3.2 and 3.3) and performs fine-tuning in order to perfect it. The input layout partially satisfies the aesthetic criteria that are of a "local" nature: blob dimension unifor-

mity between siblings (Sect. 2.1), the connection between inner structure and size (Sect. 2.2), symmetry (Sect. 2.3), blob proportions (Sect. 2.4), gap uniformity between siblings (Sect. 2.5) and space utilization (Sect. 2.6). The optimization step also tries to satisfy the global criteria: blob dimension uniformity between leaves (Sect. 2.1) and global gap uniformity (Sect. 2.5).

3.4.1 Parameter choice

The optimization step preserves the layout topology and adjusts the following parameters only:

- For areas containing two sub-areas, the division ratio of the longer edge.
- For areas containing blobs, the gaps in both directions.

Changes in these values may result in the resizing of an entire area, causing appropriate proportional resizing of its inside, and also a less significant resizing of identically sized siblings in a single area. The first kind of adjustment fixes global deficiencies that occurred at higher levels of the hierarchy tree; the second one makes local corrections.

All the optimization parameters are stored in a vector `Parameters` $= (\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_n)$. For a layout corresponding to the `Parameters` value we define `Energy` to be a measure of layout quality:

$$\mathtt{Energy} = \mathtt{GAP_WEIGHT} \times \mathtt{Gap_Var} + \mathtt{SIZE_WEIGHT} \times (\mathtt{Short_Edge_Var} + \mathtt{Long_Edge_Var}).$$

Here, `Gap_Var`, `Short_Edge_Var` and `Long_Edge_Var` are normalized variances of the set of all the gaps in the layout, and the short and the long edges of all the leaves, respectively:

$$\mathtt{Gap_Var} = \frac{\sigma[\{\mathtt{X_Gaps}, \mathtt{Y_Gaps}\}]}{\mu[\{\mathtt{X_Gaps}, \mathtt{Y_Gaps}\}]},$$

and `Short_Edge_Var` and `Long_Edge_Var` are similarly defined. μ and σ denote the mean and the standard deviation of a set. Normalization of the `Energy` function components makes them independent of the layout's actual size. `GAP_WEIGHT` and `SIZE_WEIGHT` are weighting factors with the defaults of 10 and 1, respectively (clearly, only the ratio is important).

3.4.2 Optimization process

We have implemented the classical anti-gradient walk in the solution space. Starting at the `Parameters` point corresponding to the input layout, we compute the normalized `Gradient` vector using a step size `Delta`:

$$\mathtt{Gradient} = (\mathtt{G}_1, \mathtt{G}_2, \ldots, \mathtt{G}_n),$$

where

$$\mathtt{G}_i = \frac{\frac{\partial \mathtt{Energy}(\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_n)}{\partial \mathtt{P}_i}}{\sqrt{\sum \mathtt{G}_i^2}} \approx \frac{\frac{\mathtt{Energy}(\mathtt{P}_1, \ldots, \mathtt{P}_i + \mathtt{Delta}/2, \ldots, \mathtt{P}_n) - \mathtt{Energy}(\mathtt{P}_1, \ldots, \mathtt{P}_i - \mathtt{Delta}/2, \ldots, \mathtt{P}_n)}{\mathtt{Delta}}}{\sqrt{\sum \mathtt{G}_i^2}}.$$

The `Parameters` vector is then adjusted by a `Delta` step in the direction opposite to that of the `Gradient`:

$$\mathtt{Parameters} = \mathtt{Parameters} - \mathtt{Delta} \cdot \mathtt{Gradient}.$$

Note that all the parameters are equally normalized, so that the same `Delta` can be used for all the `Parameters` coordinates.

This computation is executed in a loop until no further improvement in `Energy` can be achieved. Then `Delta` is decreased by half and the loop is rerun. In our implementation we start with `Delta` = `OPT_INIT_STEP` = 0.05, and perform `OPT_ROUNDS` = 8 decrements.

It is important to remember that this algorithm does not necessarily reach a global minimal `Energy`. It might very well get caught in a local minimum, producing a layout that is better than the initial one, but not the best one. Still, as can be seen in Fig. 18, the optimization step usually provides significant improvement of the layout aesthetics.

As discussed earlier, fully satisfying all the criteria of Sect. 2 is usually impossible, and even if we were to always find a global minimum of the `Energy` function, it would not necessarily satisfy them all. We do leave to the user, however, the option of manually running an additional optimization round, as well as that of skipping optimization entirely.

3.5 Complexity

We estimate the time complexity of the algorithm as a function of `n`, the total number of blobs, and `d`, the maximal number of siblings of a single blob (i.e., the degree of the inclusion hierarchy tree).

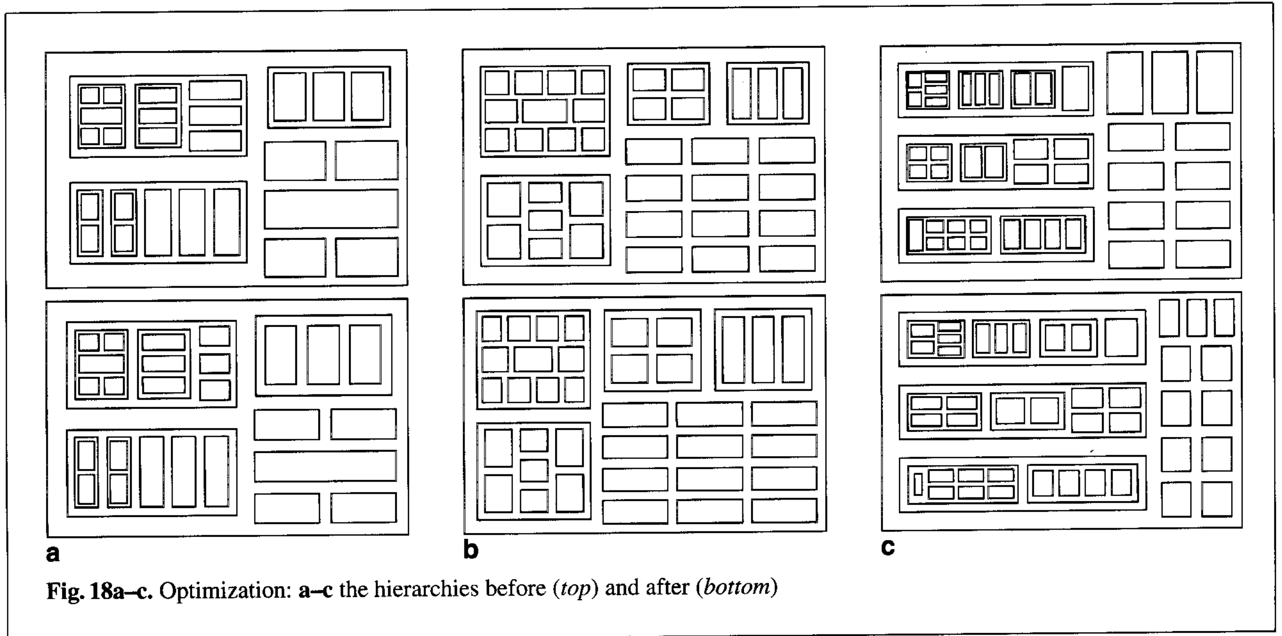


Fig. 18a–c. Optimization: **a–c** the hierarchies before (*top*) and after (*bottom*)

3.5.1 Reasonable positioning

- *Weighting*. The consumed time is that of standard depth-first search: $O(n)$.
- *Grouping*. The grouping process is carried out by each of n blobs. It involves sorting up to d offspring and a sequential pass over the ordered list. The total time is $O(n \times d \times \log(d))$.
- *Partitioning*. Each of the n blobs recursively partitions up to d groups of offspring; recursion depth is $O(\log(d))$. At each recursion level, all the offspring's sub-lists are considered sequentially by the greedy algorithm. The total time is as before: $O(n \times d \times \log(d))$.
- *Grid dimension choice*. The grid dimension is chosen by each of the $O(n)$ leaf areas. The cardinality of each area is $O(d)$. For each leaf area, $O(\sqrt{d})$ possible grids are checked, by a constant time quality estimation. The total time is thus $O(n \times \sqrt{d})$.
- *Positioning*. Each of the $O(n)$ leaf areas applies a recursive algorithm to an $O(d) \times O(d)$ grid. At each step, at least one of the grid dimensions is decreased by $O(1)$, so the recursion depth is $O(d)$. Each step involves a constant time decision. The total time is therefore $O(n \times d)$.
- *Gap computation*. Each of the $O(n)$ leaf areas evaluates a constant time function. The total time is $O(n)$.

Two additional steps of space utilization affect the complexity estimate:

- *Group sub-division*. Every one of the $O(n)$ leaf areas may potentially cause problems during grid dimension choice. In each such case, $O(d)$ sub-divisions of $O(d)$ area members will be checked by rerunning the grid dimension stage. The total time of partitioning and grid dimension step will now be $O(n \times d^2)$.
- *Size adjustment*. Each of n blobs optionally recomputes its coordinates in constant time. This does not change the complexity of the positioning step.

3.5.2 Optimization

As already mentioned, there are $O(n)$ leaf areas. The sub-tree of areas in each inner blob is binary; therefore the number of inner areas is also $O(n)$. The optimization process is performed with $O(n)$ variables. The energy function is evaluated by a depth-first search in the blobs and areas tree, consuming constant time for each tree node. We thus have a total of $O(n)$.

- *Gradient evaluation*. Each of the $O(n)$ components of the gradient vector requires two computations of the energy function. The total time of a single computation is $O(n^2)$.
- *Optimization round*. A single round evaluates the gradient vector and then updates the value of each

of the $O(n)$ values of the position vector, taking time $O(n)$. The total time is thus $O(n^2)$.

- *Optimization process.* The entire process takes several optimization rounds, performed with decreasing step value. The exact number of these rounds resulting in a convergence up to a given precision depends strongly on the energy function topology, the initial position, and the step decrement scheme. In the algorithm, optimization runs as long as improvement in the energy function value is observed. We noticed that the number of optimization rounds grows slowly for larger hierarchies, and seems to depend on the number of nodes by some sub-linear function.

3.5.3 Summary

Without optimization and space filling, the total time is $O(n \times d \times \log(d))$. This does not change for space fill by size adjustment, but grows to $O\left(n \times d^2\right)$ for space fill by group sub- division.

We do not have a fixed, precise estimate for the complexity of the optimization process, because of the unknown number of optimization rounds needed. However, it will be at least $\Omega(n^2)$, and could possibly reach $O(n^3)$.

4 Conclusions

We have developed and implemented an algorithm for drawing edgeless higraph-like structures. The algorithm employs many criteria for aesthetics, which we identified, formalized, refined, and tested. The resulting algorithm was implemented in a way intended for research and demonstration. With some adjustment, the code could probably be used in a host application.

As mentioned in Sect. 1.1, the layout of higraph-like structures has many applications but is completely undeveloped. Hence, many possible continuations of this work come to mind. The following directions seem to us to be particularly interesting:

- Adding edges and hyper-edges to the layout. Adding more complicated features, such as Cartesian products, to extend the layout to deal with statechart-like languages.
- Improving the optimization step. We used a straightforward anti-gradient walk algorithm that does not necessarily converge to the global minimum. It might be possible to improve both its complexity and the quality of the result.
- Extending the problem to three dimensions. This would involve specifying the interface (the ways to input and display 3D structures) and, of course, generalizing the aesthetics criteria and the algorithms.

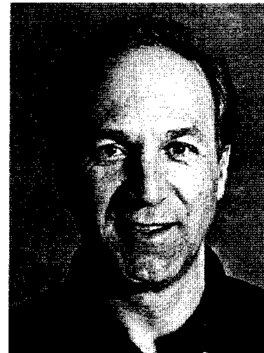
Acknowledgements. We would like to thank the anonymous referees of an earlier version of the paper for their helpful comments.

References

1. Di Battista G, Eades P, Tamassia R, Tollis IG (1999) Graph drawing: algorithms for the visualization of graphs. Prentice Hall, Englewood Cliffs
2. Davidson R, Harel D (1996) Drawing graphs nicely using simulated annealing. ACM Trans Graph 15(4):301–331
3. Eades P, Feng QW (1996) Multilevel visualization of clustered graphs. Tech. Rep. 96-09, Department of Computer Science, University of Newcastle
4. Eades P, Feng QW, Lin X, Nagamochi H (1998) Straight line drawing algorithms for hierarchical graphs and clustered graphs. Tech. Rep. 98-03, Department of Computer Science, University of Newcastle
5. Eades P, Lin T, Lin X (1993a) Two tree drawing conventions. Int J Comput Geom Appl 3:133–153
6. Eades P, Lin T, Lin X (1993b) Minimum Size h-v drawings. In: Proc. of Conf. on Advanced Visual Interfaces (AVI '92). World Ser Comput Sci 36(2):133–153
7. Eades P, Lin T, Tamassia R (1996) An algorithm for drawing a hierarchical graph. Int J Comput Geom Appl 6:145–155
8. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Prog 8:231–274
9. Harel D (1988) On visual formalisms. Comm ACM 31:514–530
10. Harel D, Gery E (1997) Executable object modeling with statecharts. Computer, July 1997, pp 31–42
11. Harel D, Lachover H, Naamad A, Pnueli A, Politi M, Sherman R, Shtull-Trauring A, Trakhtenbrot M (1990) Statemate: a working environment for the development of complex reactive systems. IEEE Trans Soft Eng 16:403–414
12. Harel D, Politi M (1998) Modeling reactive systems with statecharts: the statemate approach. McGraw-Hill, New York
13. OMG (1999) Documents on the Unified Modeling Language (UML), available from the Object Management Group (OMG), available at http://www.omg.org
14. Kandogan E, Shneiderman B (1996) Elastic windows: improved spatial layout and rapid multiple window operations. Proc. Conf. on Advanced Visual Interfaces '96, ACM Press, New York, NY, pp 29–38
15. Kant G, Liotta G, Tamassia R, Tollis IG (1996) Visibility representation of trees. Tech. Rep. CS-96-23, Center for Geometric Computing, Dept. of Computer Science, Brown University
16. Metaxas PT, Pantziou GE, Symvionis A (1994) Parallel h-v drawings of binary trees. In: Proc. 5th Int. Symp. on Algorithms and Computation (ISAAC '94), Lecture Notes in Computer Science, Vol. 834, Springer, Berlin, pp 487–496

17. Maimone MW, Tygar JD, Wing JM (1990) Formal semantics for visual specification of security. In: Visual Languages and Visual Programming, Plenum Press, New York, pp 54–64
18. Shneiderman B (1990) Tree visualization with tree-maps: a 2-D space-filling approach. ACM Trans Graph 11(1):92–99
19. Sigiyama K, Tagawa S, Toda M (1981) Methods for visual understanding of hierarchical system structures. IEEE Trans Syst Man Cybern SMC-11:109–125
20. Turo D (1994) Hierarchical visualization with treemaps: making sense of pro basketball data. ACM CHI '94 Conference Companion, pp 441–442
21. Turo D, Johnson B (1992) Improving the visualization of hierarchies with treemaps: design issues and experimentation. In: Proc. IEEE Visualization '92, IEEE Computer Society Press, Boston, MA, pp 124–131

GREGORY YASHCHIN received his M.Sc. in Computer Science from the Weizmann Institute of Science, Israel, and his B.Sc. in Mathematics and Computer Science from Tel Aviv University, Israel. His research in the field of algorithms for drawing blob hierarchies towards the M.Sc. resulted in this article. He now holds the position of project manager at Ness Technologies Inc., Israel, specializing in the field of real-time and embedded systems.



DAVID HAREL is the Dean of the Faculty of Mathematics and Computer Science at The Weizmann Institute of Science, Israel. He is also cofounder of I-Logix, Inc. He is the inventor of statecharts, the main medium for modeling behavior in many system development methods, including the UML, and was among the creators of STATEMATE and RHAPSODY. His research interests are in computability and complexity theory, logics of programs, automata theory, visual languages, systems engineering, clustering algorithms, and, more recently, the mathematics and algorithmics of olfaction and smell communication. He has published widely on these topics.

The Visual Computer

Acquiring, stitching and blending diffuse appearance attributes on 3D models

C. Rocchini, P. Cignoni,
C. Montani, R. Scopigno

Istituto di Scienza e Tecnologie dell'Informazione – Consiglio Nazionale delle Ricerche, Loc. S. Cataldo, 56100 Pisa, Italy
E-mail: roberto.scopigno@cnuce.cnr.it

Published online: 23 April 2002
© Springer-Verlag 2002

A new system for the construction of highly realistic models of real free-form 3D objects is proposed, based on the integration of several techniques (automatic 3D scanning, inverse illumination, inverse texture-mapping and textured 3D graphics). Our system improves the quality of a 3D model (e.g. acquired with a range scanning device) by adding color detail and, if required, high-frequency shape detail. Detail is obtained by processing a set of digital photographs of the object. This is carried out by performing several subtasks: to compute camera calibration and position, to remove illumination effects obtaining both illumination-invariant reflectance properties and a high-resolution surface normal field, and finally to blend and stitch the acquired detail on the triangle mesh via standard texture mapping. In particular, the smooth join between different images that map on adjacent sections of the surface is obtained by applying an accurate piecewise local registration of the original images and by blending textures. For each mesh face which is on the adjacency border between different observed images, a corresponding triangular texture patch can also be resampled as a weighted blend of the corresponding adjacent image sections. Examples of the results obtained with sample works of art are presented and discussed.

Key words: 3D scanning – Image processing – Inverse illumination – Texture_to_geometry registration – Texture mapping

Correspondence to: R. Scopigno

1 Introduction

Many applications require an accurate digital representation of both the shape and the surface attributes (e.g. the color) of an object of interest. An example is the representation and visualization of Cultural Heritage artifacts, which often require such high accuracy that the standard CAD modeling approach cannot be adopted (Levoy et al. 2000). The ideal situation would be to have an automatic 3D digital reproduction system capable of quickly and inexpensively building a very accurate and high-resolution model from the real artifact (hereafter, we assume that shape is represented by a standard triangle mesh encoding). Unfortunately, this only partially applies to the current situation. Among the many automatic acquisition technologies proposed (Petrov et al. 1998), optical range scanners provide the accuracy required by highly demanding applications and are fortunately becoming slightly cheaper, but performing a 3D scan of a complex object is still neither simple nor fast. Models with a complex topology are generally acquired with multiple scans, which have to be integrated into a single mesh (Levoy et al. 2000). The resolution of the acquisition device directly determines the complexity of the triangle mesh or points cloud produced. Mesh simplification tools (Garland and Heckbert 1997; Cignoni et al. 1999b) are generally adopted to reduce the shape redundancy of the output produced, and to support simple management of the models even on a portable computer. One critical question is how to acquire the *surface attributes* and how to link them to the shape description (Baribeau et al. 1992; Kay and Caelli 1994; Lu and Little 1995; Debevec et al. 1996; Sato and Ikeuchi 1996; Soucy et al. 1996; Sato et al. 1997; Pulli et al. 1997; Rushmeier et al. 1998; Yu and Malik 1998; Marshner 1998). The term "surface attributes" can be used to represent different concepts: from the color texture observed on the object surface, which very much depends on the lighting conditions, to the illumination-invariant surface reflectance properties (also called *albedo*), computed by removing highlights and diffuse shading or even by computing a bidirectional reflectance distribution function (BRDF). In this paper we focus mainly on the acquisition and management of surface attributes, and in particular on how we can acquire, blend and map on a standard textured mesh the detail contained in a set of images taken from different viewpoints (see Fig. 1).