Algorithms for the automatic layout of multiple windows: Practical implementations for TIDE

Hofer Nataniel, supervised by Raphaël Dumusc, Stefan Eilemann and Pr. Felix Schürmann
At Ecole Polytechnique Fédérale de Lausanne
Neurocomputing semester project

1. INTRODUCTION

Tide is an user-interface project for running large display on multi-screen. At Campus Biotech, in the Blue Brain Project premises, there is three display wall. Two of them are composed of 4x3 screens of ....pixels. The other one is 3x3. Tide is distributed and runs on any number of machines. It is the only open-source software for touch screen which scales to any size of display. At BBP it is mainly used for scientific collaboration since it makes presentations more interactive than powerpoints. The display wall allows high-resolution content, it enables pdf and movie support, it has a desktop streaming feature. With the impeding arrival of an open-deck, a 3D stereo feature has been added to add immersiveness.
Tide runs three different display mode:
-Default: where you can select windows, resize them and move them around
-Fullscreen: where the one selected image will occupy the biggest space available while keeping its ratio
-Focused: where all the selected images are displayed on one line according to the order on default mode, projecting them to fhe foreground. The image are then resized to keep their original ratio.
The presented work focuses on the former. The previous algorithm was unsatisfying because when more than a few windows were displayed, the constraint on the ratio forced a small height on the windows, thus making the content difficult to see, while most of the space was unused.
This papers explore various algorithms with different approaches to address the automatic layout problem.

2. MODELIZING THE PROBLEM AND RELATED WORK

As previously said, the ancient algorithm was unsatisfying. This leads to the following: what makes an algorithm good for this task ? The answer is not trivial as multiple criterions must be taken into account. Some of those criterions may not be easy to describe mathematically. However there is three constant rules : the original ratio must be preserved, all the windows should fit on the screen and windows must not overlap. Those constraints are easy to satisfy and formulate.
The problem itself is not clearly defined. We would like a result "pleasing to the human eye" or close to what a human may do manually. But the human evaluation of the visual aspect is subjective and hard to expain to a computer. We avoid complications by splitting the visual aspect into subcriterions:
-The windows are balanced or centered
-The content of every window is visible
-There is not unused space without reason
Multiple ways to tackle this problem will be discussed, each one having pros and cons. Most of them will lack analysis of result since implementation time was limited.

2.1 As a classical optimization problem

2.1.1 *Finding a feasible solution.* We denote l_i, r_i, t_i, b_i, being respectively the coordinates of : the left edge, the right edge, the top edge and the bottom edge of the i^{th} window (x axis grows right, y axis grows down, origin point(0,0) is set at the topleft corner of the screen). Those are our output variables. Our input values are : W and H, the width and the height of the available space, n the number of windows and R_i the ratio of the i_{th} window. Then we can write the former constraints as :

$$\frac{r_i - l_i}{b_i - t_i} = R_i \quad \forall i \in \{1, ...n\} \tag{1}$$

$$\begin{aligned} \forall i \in \{1, ..., n\} :& \\ l_i \geq 0 \land t_i \geq 0& \\ r_i < W& \\ b_i < H& \\ l_i < r_i \land t_i < b_i& \end{aligned} \tag{2}$$

$$\begin{aligned} \forall i, j \in \{1, ..., n\}, i < j :& \\ (t_i > b_j) \lor (b_i < t_j) \lor (l_i > r_j) \lor (r_i < l_j)& \end{aligned} \tag{3}$$

2.1.2 *Tuning an objective function.* A constraint solver can easily find a feasible solution. However this solution may not be satisfying, all the windows may be too small or not centered. What we precisely want is not clear, however we can turn this problem into a quadratic optimization problem with a few trick. We set the objective function f_0 to $\sum_{i=1}^{n}(r_i - l_i)(b_i - t_i)$ and we try to maximize it. To get rid of the logical constraint in equation (3), we use the method from
We introduce a constant M, big enough such that every constraint is satisfied if we put it to the right-hand side of an equality. Then we add binary variables w_m, and add to the right handside of the equation either $(1 - w_m)M$ either $w_m(M)$. Thus inequalities in (3) become :

$$\begin{aligned} \forall i, j \in \{1, ..., n\}, i < j :& \\ b_j - t_i < w_m M + w_{m+1} M& \\ b_i - t_j < (1 - w_m)M + w_{m+1} M& \\ r_j - l_i < w_m M + (1 - w_{m+1})M& \\ r_i - l_j < (1 - w_m)M + (1 - w_{m+1})M& \end{aligned} \tag{4}$$

Thus, our problem becomes almost a quadratic programming instance (because of the integer binary variables it is not). We can bypass this issue by assigning first the binary variables and then solv-

ing the quadratic programming problem using Lagrange multipliers. Complexity explodes since there is $O(n^2)$ variable, so $O(2^{n^2})$ assignments possible. In practice, for Tide, n would rarely exceed 20. Brute force may be a viable solution. If computation time is slow, assigning randomly those variable and solving the quadratic program may be a better alternative.
At this point, we are guaranteed to have a satisfying occupancy of the space. But we still have the issue that window can have close to 0 width. There are two possible workaround :
-Add constraints to enforce a min size for the windows.
-Add a penalizing term to the objective function.
Depending on the min size constraint, alternative 1 may lead to unfeasible solution. Running the algorithm for different values of min size will lead to a satisfiable solution, with a computational cost.
The difficulty in alternative 2 is to find a function which penalizes smallest and biggest windows. We suggest to add $\lambda \sum_{i=1}^{n} w_i + h_i$ to the function f_0 where $w_i = r_i - l_i$ and $h_i = t_i - b_i$ and λ is a real that must be tuned appropriately.
Our algorithm has improved but is still unsatisfying since there is no guarantee of equilibrium. Similar input conditions may lead to very different results. Implementation of this algorithm has been put aside to the profit of more heuristic ones, allowing a greater control over the output.

3. HEURISTIC ALGORITHMS

3.1 Binary tree algortihms

3.1.1 *Main principles.* Since we convinced ourselves that an optimal solution may be hard to compute, we thought about finding an algorithm which gave good results on most of the cases without any "hard" guarantees. The idea is to construct a rectangle containing all the windows with ratio as close as possible to the available space. We start with an empty tree and add all windows one by one. Once there is no remaining window, we resize the tree so it fits in the available space. For this algorithm to perform well, we need to keep a rootNode whose rectangle is similar to the available space (meaning their ratio width/height is close).

Any node in this tree is a rectangle composed of two children which are contiguous rectangles. A leaf that contains a window is called "full" and a leaf that does not contain a window is "empty".

The insert procedure is easy to implement. We try inserting recursively in the tree, searching for an empty leaf big enough. If such insert was unsucessful, we create a new rootNode containing the new window and the previous tree as children. The position of the new window is chosen so the tree is closest to the objective ratio. (see Figure 1, 2 and 3 for an example)

Once the inserting part is done, our tree (or rootNode since it is equivalent) may be way bigger than our available space. Or, to the contrary, very small. This does not matter since we resize our tree to fit in the available space. This constraining phase is also easy to implement recursively on our tree, since each node is a rectangle whose space is the sum of his children. When leaves are constrained in a rectangle, they take the biggest space possible without changing their original ratio and are then centered.

3.1.2 *Initialization choices.* The attentive reader has noted two unspoken initialization processes which affect the execution of the algorithm. We did not discuss the initial size of the windows and the order in which the algorithm inserts them.
In Default mode of tide, windows have a size when they are opened depending on their content. Choice has been made to use this size as initial size of the windows. By the way the algorithm works,

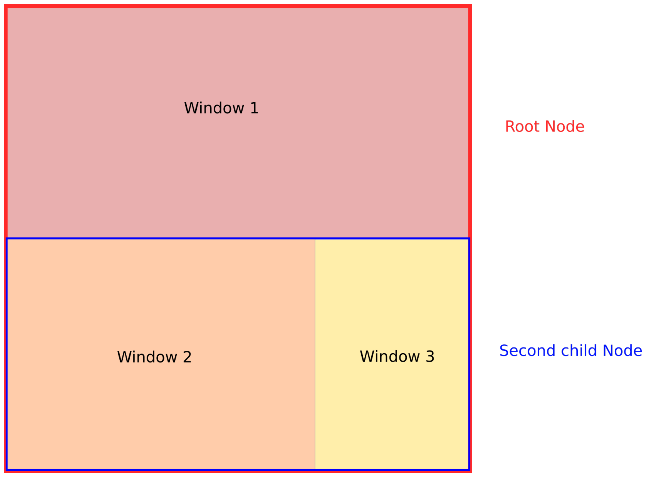

Fig. 1. An example of insertion of 3 windows into an available space of ratio 1

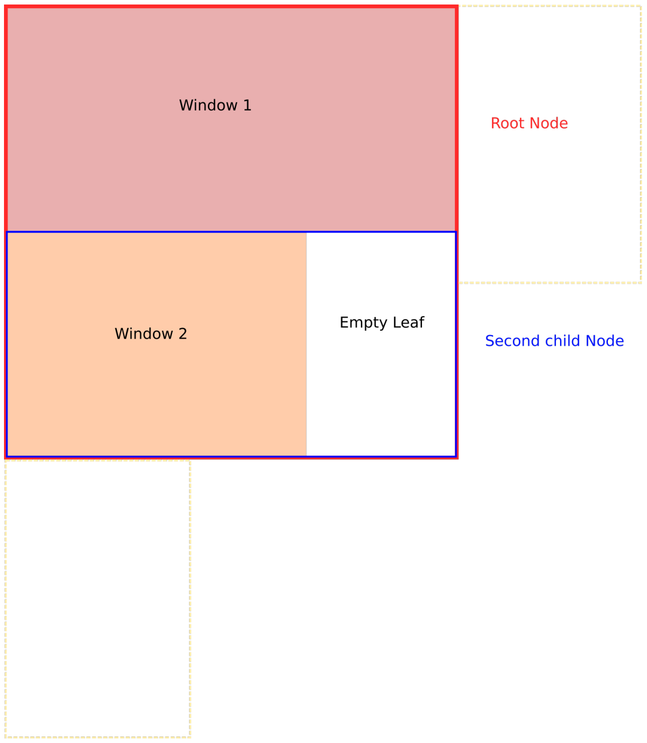

Fig. 2. This time the empty leaf is too small, we choose emplacement to the right because of the ratio = 1

the relative size of two windows on the focused mode is the same than in default mode. This is a benefit in most of the cases, where default mode has already a balance between the windows. But it leads to degenerate cases when a window occupies the majority of the screen in default mode. Such cases can be detected and an initial resizing of the window may occur. An other benefit resulting from this initialization is the control over the algorithm brought to the user. He can decide on default mode to increase the size of a window and this will impact its size in focused mode. At the

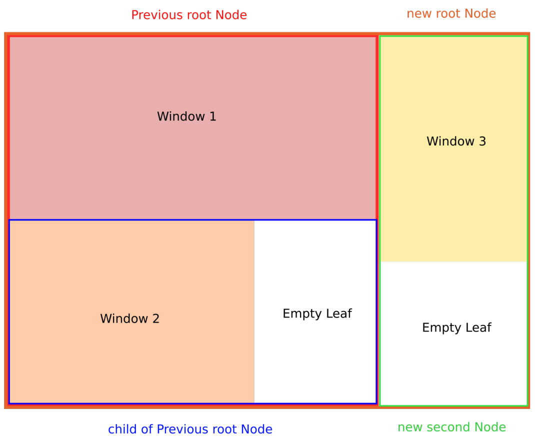

Fig. 3. The structure of the tree has been changed

moment it is the only control the user has on the display of this algorithm. In the previous focus mode, control offered to the user laid in the order (left from right) of the windows.
This is not the case anymore with this algorithm since the order of the windows depend on the maximum of their relative width or relative height compared to the available space. Those "bigger" windows may indeed prove troublesome if they arrive last since they are more susceptible to change the ratio of the tree. However, in some cases this ordering is far from the best one. We will see a way to deal with this issue.

3.1.3 *Weaknesses in the algorithm.* One of the main weakness in the algorithm lies in the representation of empty spaces. A window may fail to insert into two contiguous empty spaces because they lie far from each other in the binary tree representation.
This issue is the most troublesome for this algorithm as it is intrinsically connected with the binary tree representation which is a key feature for the resizing part. We could imagine of a way to detect close empty spaces and considering them as one big empty space, making some insertion to rebalance the tree. This feature as not been implemented yet, because it would lead to fundamental changes in the algorithm. Those unconnected problems were first addressed by changing the underlying data structure from a binary tree to a grid of cuts where each window is defined by four cuts, two by two parallel. The insertion part led to better results before the resizing, however the resizing part was far from trivial and may be as difficult as the initial problem.

Another flaw is the lack of flexibility when we insert a new window. As we can see in the following example where 4 images of the same size are selected, due to floating points error. The last image is then inserted to the left instead of having a nice 2 by 2 square.
A workaround is to add a tolerance factor when inserting a new window. If the window height and width is no more than $1 + \alpha$ times the empty space, the window is added to the empty space shrinking it if need be. There is a tradeoff for the value of α. If it is zero, we lack flexibility as in figure If it si high, the relative ratio between windows is not respected anymore. Current value of α is 0.3, it has been determined empirically.

Now we have come to the order issue. While sorting by ratio may often be a good idea, there are always corner cases in which it may be the worst. In particular when a window is big and its ratio is far from the others.

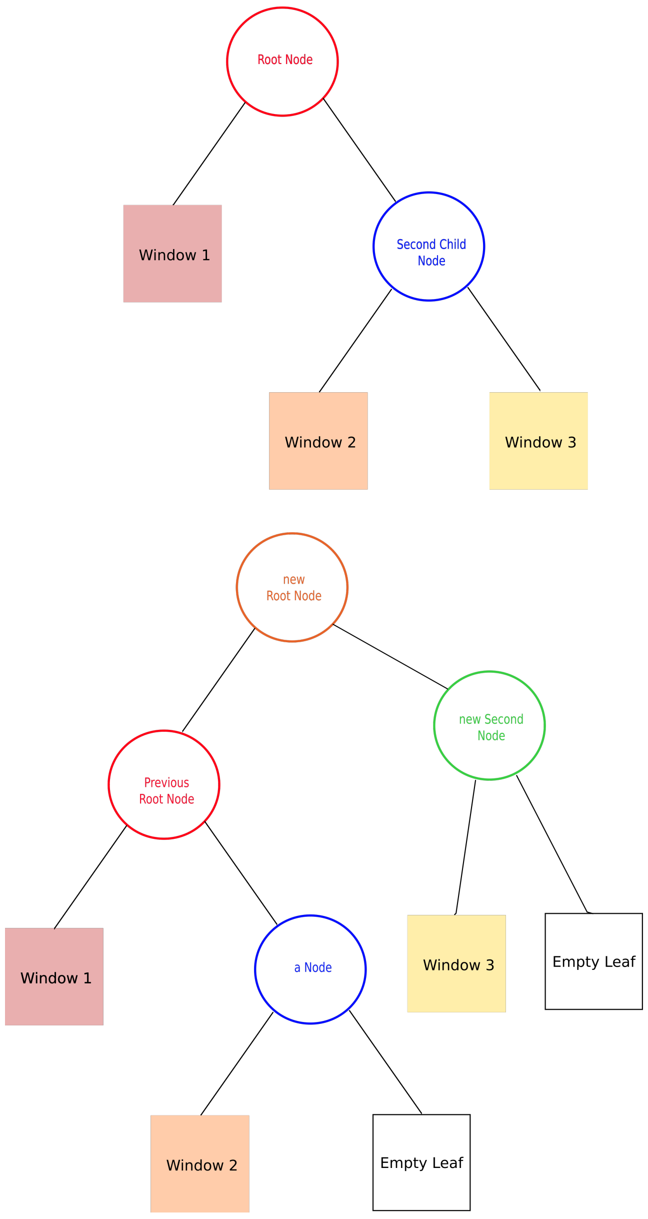

Fig. 4. comparison between the tree structure of Figure 1 (top) and Figure 3 (bottom)

A natural way to deal with those cases is to randomize the order of the windows and run the algorithm multiple times. We keep the result for which the occupied space is maximal. Note that brute force requires $O(n!)$ runs and is thus not a viable solution. A specific study of the number of iterations needed in expectation to come close to the optimal result has not been established. It has been noticed that this method improved the results compared to the original order. The main issue is that the output is not determined by the input, which can be destabilizing for the user. If the user does not like this, it is always possible to fix the seed of the random function. The same input will always produce the same output, which can be an issue on specific inputs. However there is no reason that someone would "cook up" especially bad inputs, and thus having a known pseudo-random function instead of a unfathomable is acceptable.

3.2 Performance Analysis

ACKNOWLEDGMENTS