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

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

The present paper focuses on the problem of automatic layout of multiple windows. We consider two different approaches. The first one is to bring the problem to a known global optimization program. The second method used consists at finding an heuristic algorithm which optimizes the visual result in multiple steps.

■

1. INTRODUCTION

Tide is a user-interface project for running large display on multi-screen. At Campus Biotech, in the Blue Brain Project (BBP) premises, there are three display wall. Two of them are composed of 4x3 screens of 1920x1080 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 allows for more interactive presentations than powerpoints. The display wall allows high-resolution content, it enables pdf and movie support, and has a desktop streaming feature. With the impeding arrival of an open-deck, a 3D stereo feature has been added to improve immersiveness.
Tide runs three different display mode:
-Default: where you can select windows, resize them and move them around;
-Fullscreen: where the selected image will occupy most of the space available while keeping its aspect ratio;
-Focused: where all the selected images are displayed on one line according to the order on default mode, projecting them to the foreground. The image are then resized to keep their original aspect ratio.
The presented work focuses on the later. The previous algorithm was unsatisfying when more than a few windows were displayed, because most of the space was unused. The single line display forced a small height on the windows, thus making the content difficult to see.
This papers explores two different approaches to address the automatic layout problem.

2. MODELIZING THE PROBLEM AND RELATED WORK

2.1 Different perspectives on the subject

Layout problems are often NP-hard, algorithms used to address this kind of problem use heuristic (Balakrishnan et al. 2003 [1]. Luders et al. (1995) [2] used combinatorial optimization, which differ from the methods in the present paper.
A first part will show how to consider the problem as multiple quadratic programming instances. SImilar optimization problems can be found in different domains : architecture (Michalek et al. 2002 [3]), economics (Koopmans and Beckmann 57 [4]) and microengineering (Liebmann et al. 2003 [5]).
The second part will focus on an heuristic algorithm implemented to resolve this task.

2.2 Requirements

As previously stated, the previous algorithm was unsatisfying, because of the visibility of the content and the usage of available space. We must find all characteristics an algorithm should have for this task to be satisfying. The answer is not trivial as multiple criterions must be taken into account. Some of those criteria may not be easy to describe mathematically. However there are 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 the result to be “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 subcriteria:
-Windows keep their aspect ratio
-Windows fit in the available space
-Windows do not overlap
-The windows are regularly spaced
-Unused space is minimal
-Variance in area is minimal
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.

2.3 As a global optimization problem

2.3.1 *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. For the sake of readability we introduce the following notation $w_i = r_i - l_i$ and $h_i = t_i - b_i$ which are the width and the height of window i. Then we can write most the former constraints as :

$$\frac{w_i}{h_i} = R_i \quad \forall i \in \{1, ...n\} \tag{1}$$

Those equalities capture the constraint on the aspect ratio of each window.

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

Those inequalities ensure that the windows fit in the available space.

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

This last equation is a bit more tricky. Here we ensure that for each pair of windows (i,j) : j is either ,under, above, to the right or to the left of i. That is exactly the definition of "not overlapping". Note that we use $i < j$ because "not overlapping" is a symmetrical relation. E.g. if window 2 is to the left of window 5, then window 5 is to the right of window 2. Thus there is no need to enforce that constraint.
A constraint solver can easily find a solution to this problem.

2.3.2 *Turning the problem to a quadratic programming instance.* Now we have a feasible solution. However this solution may not be satisfying since there is no constraint on the minimization of the unused space. Since the barrier between optimization and constraint programming is thin (Hooker 2002 [6]) , we can turn this problem into an optimization problem by setting the objective function to :

$$f_0(n) = max \sum_{i=1}^{n} (w_i)(h_i) \tag{4}$$

The goal is now to go back to a quadratic programming instance, i.e to write the problem as :

$$\begin{aligned}\text{maximize} \quad & f_0(x)\\ \text{subject to} \quad & f_i(x) \leq 0 \quad \text{for i = 1,...,m}\\ & f_j(x) = 0 \quad \text{for j = m + 1, ..., q}\end{aligned} \tag{5}$$

Where x is a vector of real numbers, $f_0(x)$ is a quadratic function of x, $f_i(x)$ and $f_j(x)$ are linear function of x.
Our output variables form vector x. Our objective function is indeed quadratic in x, the inequalities in (2) are trivially linear.
We can write the equalities in (1) as :

$$w_i - R_i h_i = 0 \quad \forall i \in \{1, ...n\} \tag{6}$$

And that is also trivially linear in x.

The trouble lies with the logical constraint in equation (3). To get rid of them we use the method from this excellent course of the MIT.
We introduce a positive constant M, big enough such that every constraint is satisfied if we put it to the right-hand side of an equality (W+H for example). Then we add binary variables v_m, and add to the right-hand side of the equation either $(1-v_m)M$ or $v_m(M)$. Thus inequalities in (3) become :

$$\begin{aligned}&\forall i, j \in \{1,...,n\}, i < j :\\ &b_j - t_i < v_m M + v_{m+1} M\\ &b_i - t_j < (1 - v_m)M + v_{m+1} M\\ &r_j - l_i < v_m M + (1 - v_{m+1})M\\ &r_i - l_j < (1 - v_m)M + (1 - v_{m+1})M\end{aligned} \tag{7}$$

With :

$$m = 2 * (j + \sum_{t=1}^{i} n + 1 - i)$$

Fix i and j, whatever the assignment of the variables v_m and v_{m+1} the right hand-side of one in the above four inequalities will be 0 and thus the not overlapping constraint will be met. The other three inequalities will be trivially satisfied. E.g . $v_m = 0$ and $v_{m+1} = 0$ enforce that window j is above window i. There are two different variables v for each of these relation, hence $n * n + 1$ variables v. Let us write v the vector containing all the variable v_m. (x,v) is a feasible solution to our new problem only if x is a feasible solution to the previous one. Moreover if x is feasible, then there exists a v such that (x,v) is feasible.
However there exists assignments of v such that our problem is not satisfiable anymore. E.g. window 1 above window 2, window 2 above 3 and window 3 above 1.
Thus, our problem becomes a quadratic programming instance for every assignment of v. We know how to solve them using Lagrange multipliers (Frank et al. 1956 [7]). Complexity explodes since there are $O(n^2)$ variables, so $O(2^{n^2})$ assignments possible. In practice, for Tide, n would rarely exceed 20. Hence brute force may be a viable solution. If computation is slow, assigning randomly those variable and solving the quadratic program may be a better alternative. It will probably work although many assignments of v lead to unfeasible problems because many other assignments of v will work and will be equivalent in a way.

2.3.3 *Minimizing variance in the windows.* At this point, we are guaranteed to have a satisfying occupancy of the space. But we still have the issue that windows 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, the first alternative may lead to an empty feasible set. Running the algorithm for different values of min size will lead to a satisfiable solution, with an increase in computational cost.
The difficulty in the second alternative is to find a function which penalizes smallest and biggest windows. We suggest to add to the objective function (4) the following term :

$$\lambda \sum_{i=1}^{n} w_i + h_i \tag{8}$$

Where λ is a real that must be tuned appropriately. Note that this idea derives from the classical regularization of least-square problem (lasso regression) where we try to minimize $||Ax-b||_2 + ||x||_1$
Our algorithm has improved and we think it could be interesting to implement it. However since we first thought it would be hard to reach a satisfiable result in a reasonable amount of time we decided to focus on another algorithm.

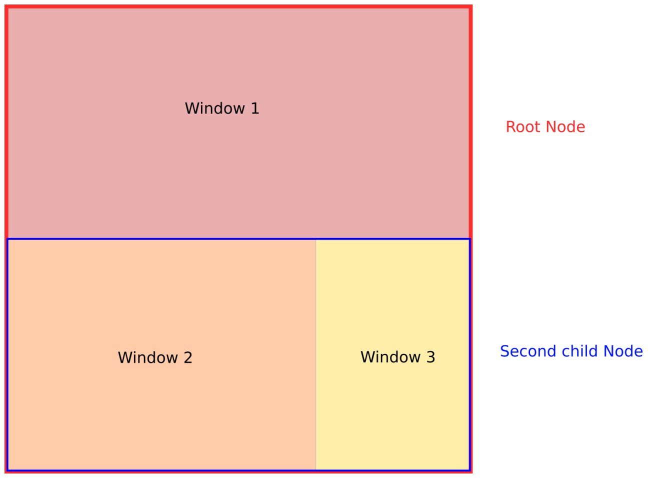

Fig. 1. Insertion of 3 windows into an available space of ratio 1

3. HEURISTIC ALGORITHM

3.1 Introduction

Since we convinced ourselves that an optimal solution may be hard to compute, we looked for an algorithm which gave good results in most of the cases without any "hard" guarantees.

3.1.1 *Main principles.* 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 the windows one by one. Once there is no remaining window, we resize the tree to fit in the available space. For this algorithm to perform well, we need to keep a rootNode whose surface is similar to the available space (meaning their ratio width/height are close).

Any node in this tree is a rectangle composed of two children which are contiguous. 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 a big enough empty leaf . If such insert is 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, on 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 its 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 may have 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 default size when they are opened (depending on their content). The choice has been made to use this size as the initial size of the windows. By the way the algorithm works, 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 windows in Default mode are already balanced. But it leads to degenerate cases when a window occupies the majority of the screen in Default mode. However such cases could be detected and we could consider resizing the window before running the algorithm. This feature is actually not implemented.

Another 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 moment it is the only control the user has on the display of this algorithm. In the previous focus mode, control of-

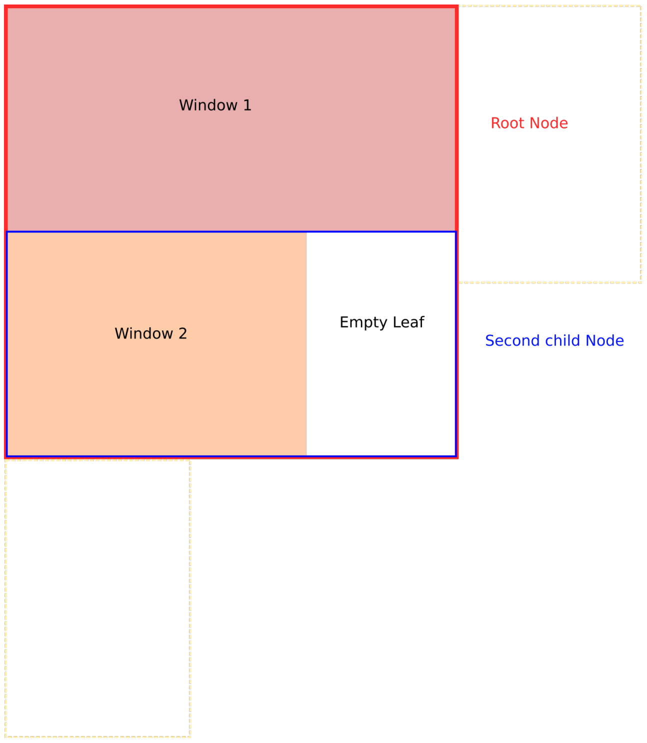

Fig. 2. This time the empty leaf is too small, we choose emplacement to the right because of the ratio = 1

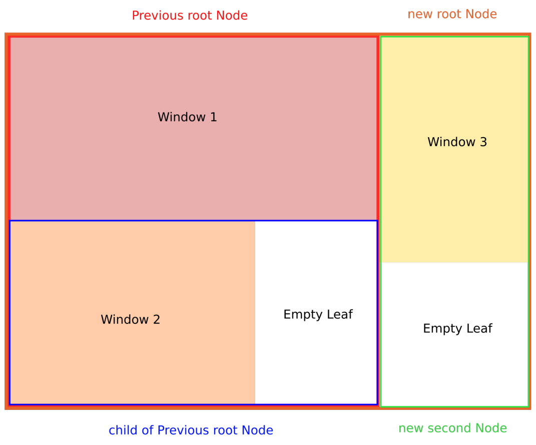

Fig. 3. The structure of the tree has been changed

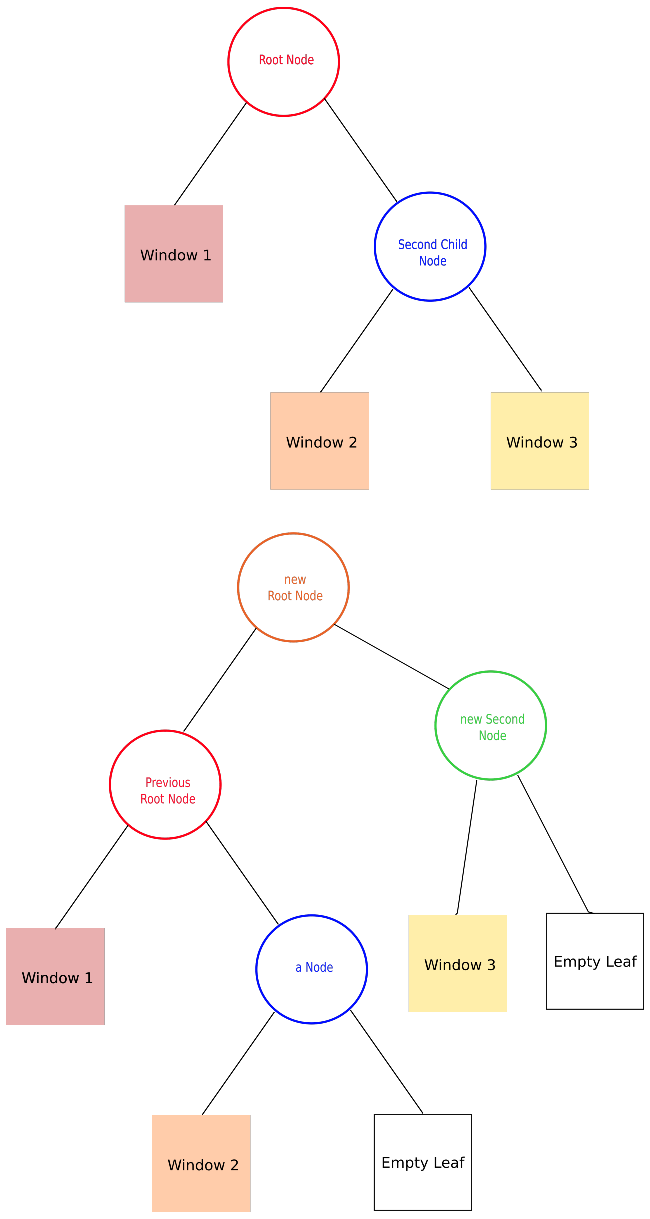

Fig. 4. comparison between the tree structure of Figure 1 (top) and Figure 3 (bottom)

fered 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 depends 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.

3.2 Issues and workarounds in the algorithm

3.2.1 *Representation of empty spaces.* 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 a way to detect close empty spaces and considering them as one big empty space, making some insertion to rebalance the tree. This feature has not been implemented yet because it would lead to fundamental changes in the algorithm. We attempted to address this problem 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. Thus, the idea of cuts has been put aside.

3.2.2 *Tolerance factor.* Another flaw is the lack of flexibility when we insert a new window, as we can see in the Figure 5 where four images of the same size are selected. The last image is then inserted to the left instead of having a nice two by two square, due to floating points error.
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 still 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 5. If it is high, the relative size ratio between windows is not respected anymore. Empirically deduced value of α is 0.3. See Figure 6 for an example with tolerance factor.

3.2.3 *Order of the windows.* While sorting by aspect ratio may often be a good idea, there are always corner cases in which it may be the worst order possible. Such cases arise when a window is big and its ratio is far from the others.

A natural way to deal with those cases is to randomize the order of the windows and run the algorithm multiple times, keeping the result for which the occupied space is maximal. Note that brute force requires $O(n!)$ runs and thus is not a viable solution. A specific study of the number of iterations needed 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 an unfathomable one is acceptable.

3.2.4 *insertion phase using recursivity.* While the improvement randomization brought on the special cases was substantial, the solution it led to was not perfect. The heuristic placement in insertion phase leads to a sparse rectangle. The resizing is then unable to balance the windows in a nice way. Considering each subtree as the same optimization problem with another available space during insertion phase might fix some of those problems. Thus, each node would have a different available space which will not be equal to only the sum of his two childs. According less importance to the rootNode and allowing each node to modify the structure of the tree would add the flexibilty we lack. However this feature has not yet been implemented.

3.3 Results

Although the quality of the algorithm was not assessed formally but more on visual aspects during implementation, we show here

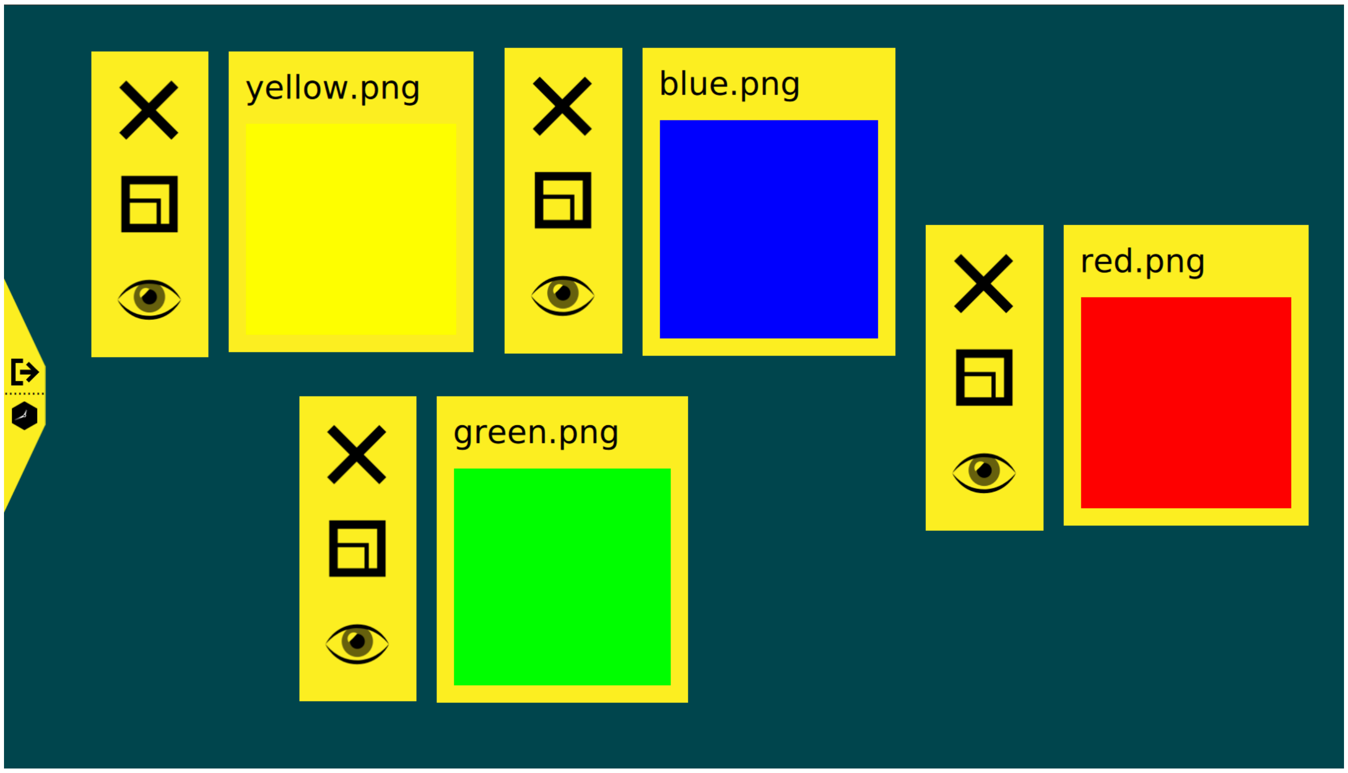

Fig. 5. A two by two display here would be nicer

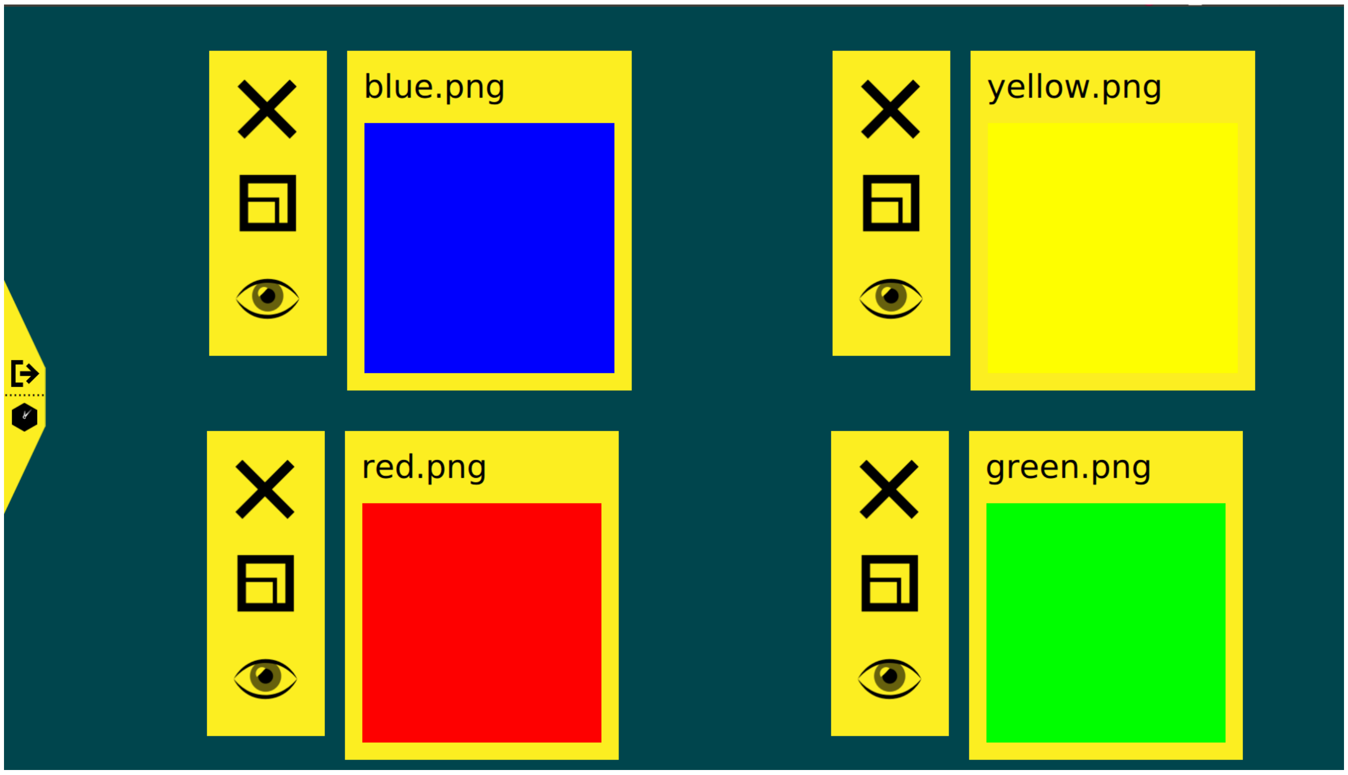

Fig. 6. With tolerance factor, we reach a two by two display

Table I. Percentage of occupied space by windows on different sets

Algorithm used	Low Variance	High Variance	Mixed
Previous algorithm	9.95 %	11.08%	2.40%
No tolerance	61.22%	51.81%	51.60%
1.3 tolerance factor	61.22 %	41.94%	47.24 %
20 random shuffle	61.89 %	69.50%	61.88 %
200 random shuffle	63.90 %	69.32 %	66.50 %

an analysis on different set of windows. Three sets of windows were used. The first one consisted of six windows with high variance in height, width and aspect ratio. The second one had low variance in those parameters while the third one was simply the union of the first two.
We used the percentage of space occupied by all the windows relative to the available space as a metric. The previous algorithm along with four different version of the new heuristic one were compared. The first version presented no tolerance factor during insertion and no permutation in the order of insertion of the windows. The second one presented in addition a tolerance factor of 1.3. The third and fourth one had respectively 20 and 200 random shuffles in the insertion order. The results are shown in the Table I.

The most salient result is the huge improvement in space-using done by the heuristic algorithm over the previous one. Moreover, the algorithm did not lose significant performance even when there was 12 windows (Mixed case).
Furthermore, we see that without shuffling the windows, the algorithm performed worse on the high variance set and with shuffling it can find a better solution. In low variance cases, shuffling does not improve much the solution, probably because we are already close to the optimal order for this algorithm.
Strangely the tolerance factor added seem to decrease the performance with high variance windows. Since it was clearly an improvement in other cases, I would suggest to run the algorithm with multiples values of tolerance and keep the best results as we did with randomness.

4. CONCLUSION AND FUTURE WORK

We managed to design an algorithm which performs well on most of the cases although the core idea is very simple. Compared to the previous one and also to what an human will do. A few features greatly improved the cases on which the algorithm was less performant. We presented multiple ideas to reduce further the occurences of less satisfying solutions, notably with the change of available space in each subtree insertion phase.

On a different perspective, now that the automatic layout performs well on most of the cases, focus should be put on user-interface features. We would like to give to the user the possibility to move, resize or swap some windows in focus mode without changing the result of the other windows.

We showed that we can consider this problem as multiple instances of quadratic programming. It would be interesting to implement an algorithm for this optimization. On one hand it could be more performant than the present heuristic one . On the second hand, it would be very easy to add user-driven constraints. E.g. A peculiar window is at the leftmost part of the screen, or is at fixed size.

REFERENCES

Jaydeep Balakrishnan, Chun-Hung Cheng, and Kam-Fai Wong. Facopt: A user friendly facility layout optimization system. *Computers & Operations Research*, 30(11):1625–1641, 2003.

Peter Lüders, Rolf Ernst, and Stefan Stille. An approach to automatic display layout using combinatorial optimization algorithms. *Software: Practice and Experience*, 25(11):1183–1202, 1995.

Jeremy Michalek, Ruchi Choudhary, and Panos Papalambros. Architectural layout design optimization. *Engineering optimization*, 34(5):461–484, 2002.

Tjalling C. Koopmans and Martin Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25(1):53–76, 1957.

Lars W Liebmann, Greg A Northrop, James Culp, Leon Sigal, Arnold Barish, and Carlos A Fonseca. Layout optimization at the pinnacle of optical lithography. In *Advanced Microelectronic Manufacturing*, pages 1–14. International Society for Optics and Photonics, 2003.

John N. Hooker. Logic, optimization, and constraint programming. *INFORMS Journal on Computing*, 14(4):295–321, 2002.

Marguerite Frank and Philip Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1-2):95–110, 1956.

ACKNOWLEDGMENTS