Towards optimal table layout

Nathan Hurst
School of Computer Science and Software Engineering
Monash University
Clayton, Victoria 3168, Australia
njh@mail.csse.monash.edu.au

Kim Marriott
School of Computer Science and Software Engineering
Monash University
Clayton, Victoria 3168, Australia
marriott@mail.csse.monash.edu.au

ABSTRACT

Tables are one of the most powerful and useful design elements in current web document standards such as (X)HTML, CSS and XSL. Importantly, designers do not need to precisely specify the width of the table columns, instead the designer may allow these to adapt to the viewing context while still preserving the general design intended by the document author. Unfortunately, however, the adaptive layout provided by tables is still not powerful enough. We present a new approach to table layout that extends the current standard by allowing the designer to specify arbitrary linear equality and inequality constraints over the column widths and row heights. These may be strict (required) or preferred with weights indicating their relative importance. We also allow cells to be non-rectangular regions and we allow text to flow between arbitrary cells. We view table layout as a constrained optimisation problem. In addition to the designer's constraints there are implicit constraints reflecting the structure of the table and a constraint that each cell must be large enough to contain its content. Accurately and efficiently handling this containment constraint is the main source of difficulty. We present and evaluate two possible approaches, both of which are based on quadratic programming techniques.

1. INTRODUCTION

Tables are one of the most powerful and useful design elements in current web document standards such as (X)HTML, CSS and XSL. Indeed because of their power, tables are frequently (mis)used by web designers to finely control page layout, not just to display tabular information. Unlike the case in many document formatting systems, for example LaTeX, authors do not need to precisely specify the width of the table columns, instead the author may allow these to adapt to the viewing context while still preserving the general design intended by the author. This allows better use of the viewing device by taking into account the window size and aspect ratio and also caters for viewer needs such as larger fonts.

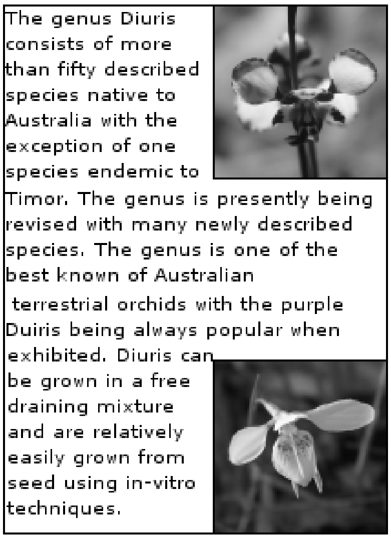
The current (X)HTML, CSS and XSL specifications provide a rather complex specification for table layout which essentially gives an algorithm for layout that works by propagating minimum and maximum sizes of cells in the table. Unfortunately, however, HTML, CSS and XSL tables are still not powerful enough for at least five reasons.

- It is not possible to specify non-rectangular compound cells in which for instance we can allow text to flow around an image. It is also not possible to specify that text flows sequentially through a number of cells, allowing for instance multicolumn layout in tables. Removing these limitations are particularly important when tables are used for page layout rather than displaying tabular data. This is a limitation that the W3C have highlighted in the draft CSS3 specification [6]. As an example of a table requiring these capabilities see Figure 1.

- Indeed the current layout algorithm does not even handle multi-paragraph text in cells or multi-column cells very well. As an example consider the two tables shown in Figures 2 and 3 which detail layout using Mozilla compared against the layout we obtain with our approach.

- The constraints on column width that may be specified by the designer are too limited and somewhat ad hoc. For instance, it is possible to specify that column widths are multiples of a single unknown width but not to specify their widths in terms of *two* unknown widths.

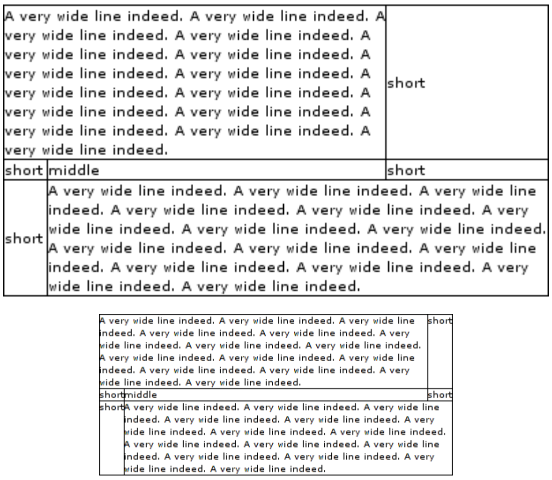
- It is not possible for the designer to give possibly conflicting design requirements with an associated strength and for the system to try and best satisfy these by minimising the overall



The genus Diuris consists of more than fifty described species native to Australia with the exception of one species endemic to Timor. The genus is presently being revised with many newly described species. The genus is one of the best known of Australian terrestrial orchids with the purple Duiris being always popular when exhibited. Diuris can be grown in a free draining mixture and are relatively easily grown from seed using in-vitro techniques.

Figure 1: A simple example of non-rectangular compound cells with text flow as might occur in brochure layout rendered using our algorithm. The width and height of the table is fixed and the image size will increase to minimise whitespace while preserving the aspect ratio of the image.

WWW2004, May 17–22, 2004, New York, NY USA.
ACM xxx.xxx.



A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed.		short
short	middle	short
short	A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed.	

A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed.		short
short	middle	short
short	A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed. A very wide line indeed.	

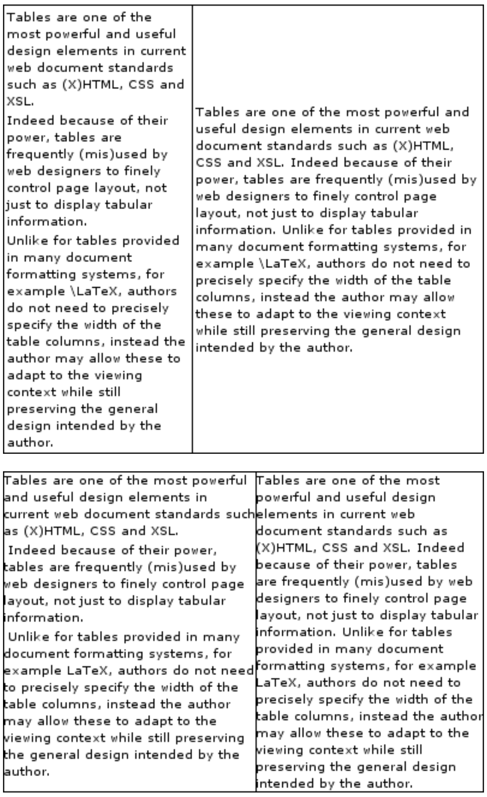
Figure 2: An example table with three colums and two cells spanning two columns rendered using Mozilla's default rules and then using the algorithm presented here.

conflict. Instead, the system will ignore the less important design constraints. For instance, the designer may want to specify that two columns should definitely be the same width and that if possible they should be 20% of the table width.

- It is not possble to specify constraints on the row heights. For instance one may want all rows to have the same height. It also is not possible to specify that a cell should have a fixed aspect ratio.
- Constraints spanning more than one table are not allowed. For instance, it is not possible to specify that columns in two different tables should have the same width[1].
- The layout algorithm implicitly encodes a single criterion for optimising table layout. The designer cannot specify a different layout criterion such as for example maximising readability or minimising table width subject to the height still being short enough to fit in the browser view window.

It is worth noting that the importance of good automatic table layout is set to increase for at least three reasons. First, it is increasingly common for document content to be generated dynamically, for instance from a database. In such cases web designers can only provide a default layout since they do not have detailed knowledge of the document content. Second, the range of devices used to view web-pages has increased enormously, and the same document needs to look good when viewed using a lap-top, PDA, or a wall-mounted video display. Third, material on the web is increasingly being used as the primary source for production quality printing etc. The requirements for print-media based layout are much more stringent than for on-line viewing. This is confounded by the different page sizes used in different countries, e.g. US letter in the US and A4 in Australia.

Here we give a new algorithm for table layout which overcomes the above limitations. Importantly, these are not ad hoc extensions but arise naturally out of our approach which is to model table layout as a constrained optimisation problem. The designer is free to specify arbitrary linear constraints over both column widths and row heights. These have an associated strength indicating how important they are to satisfy. In addition there is an implicit constraint that each cell must be large enough to contain its content.



Tables are one of the most powerful and useful design elements in current web document standards such as (X)HTML, CSS and XSL. Indeed because of their power, tables are frequently (mis)used by web designers to finely control page layout, not just to display tabular information. Unlike for tables provided in many document formatting systems, for example \LaTeX, authors do not need to precisely specify the width of the table columns, instead the author may allow these to adapt to the viewing context while still preserving the general design intended by the author.	Tables are one of the most powerful and useful design elements in current web document standards such as (X)HTML, CSS and XSL. Indeed because of their power, tables are frequently (mis)used by web designers to finely control page layout, not just to display tabular information. Unlike for tables provided in many document formatting systems, for example \LaTeX, authors do not need to precisely specify the width of the table columns, instead the author may allow these to adapt to the viewing context while still preserving the general design intended by the author.

Tables are one of the most powerful and useful design elements in current web document standards such as (X)HTML, CSS and XSL. Indeed because of their power, tables are frequently (mis)used by web designers to finely control page layout, not just to display tabular information. Unlike for tables provided in many document formatting systems, for example LaTeX, authors do not need to precisely specify the width of the table columns, instead the author may allow these to adapt to the viewing context while still preserving the general design intended by the author.	Tables are one of the most powerful and useful design elements in current web document standards such as (X)HTML, CSS and XSL. Indeed because of their power, tables are frequently (mis)used by web designers to finely control page layout, not just to display tabular information. Unlike for tables provided in many document formatting systems, for example LaTeX, authors do not need to precisely specify the width of the table columns, instead the author may allow these to adapt to the viewing context while still preserving the general design intended by the author.

Figure 3: An example table with a cell containing three paragraphs rendered using Mozilla's default rules and then using the algorithm presented here.

Accurately and efficiently handling this containment constraint is the main source of difficulty. The problem is that the cell containment constraint is inherently non-linear. Each cell has a finite number of different possible layouts corresponding to different height-width configurations. For instance, in the case of text, the different configurations correspond to different number of lines in the layout. We regard table layout as the problem of finding the best configuration for each cell in the table. That is the one which best satisfies the preferred constraints. which might encode for instance maximising the use of the viewing area and the design constraints.

Our table layout algorithm is based on linear programming techniques [7] developed in operations research for finding the solution to a system of linear arithmetic constraints which best minimizes a linear objective function and a dynamic linear approximation technique used in interactive graphical applications for modelling complex non-linear constraints by a changing collection of linear constraints [8]. In our context the objective function encodes the penalty associated with the preferred constraints. The algorithm works by staring from the widest possible configuration for the table and then choosing a column to narrow to the next configuration until further narrowing will not improve the objective. The algorithm uses the Lagrangean multiplier associated with the

[1] One can specify that they have the same fixed width or the same width relative to the page width but it is not possible to only specify that they should have the same width and then allow this width to be computed dynamically from the natural cell content widths.

column bounds to guide and terminate the search. In theory the algorithm has polynomial complexity, and in practice it is quite fast. Of course since table layout is inherently NP-hard, our algorithm is not guarranteed to find the best solution. However again in practice it seems to find very good layout. One of the main difficulties is handling non-rectangular complex cells. We use a linear constraint that approximates the relationship between the area of the cell and the width and heights of the columns and rows of the component cells.

This is not the first paper to model table layout as constrained optimisation. Wang[11] investigated the problem of semantic modelling of tables and presented a branch and bound algorithm, accelerated with a polynomial greedy algorithm. Anderson and Sobti [1] provide further analysis of the complexity of the table layout problem and provide some additional work on quickly generating a list of potential text sizes. The most closely related work is that of Borning et al [4, 5, 2] who allowed the designer to specify required and preferred linear arithmetic constraints over column widths. However none of the previous work handles non-rectangular compound cells or constraints such as fixed aspect ratio which relate table height with table width. These are the main source of difficulty in the algorithm.

In Section 2 we describe our model for table layout as constrained optimisation. In Section 3 we describe our algorithm for table layout and in Section 4 we provide a preliminary evaluation. Section 5 summarises our contributions and future work.

2. THE MODEL

HTML 4.0, CSS and XSL all provide a very similar table formalism. For convenience we shall refer to this as the *standard table model*. In this model the designer can specify that a column has an absolute width such as 1 inch or a width relative to the width of the surrounding object, for instance specifying that a column is 20% of the page width. They can also specify the width of the table in this way. We say columns with an absolute or relative width have *fixed-width*. The designer can also specify that a column should have width that is some ratio of the special width "$\star$".

Each cell c has a minimum width $c.minwidth$ which is the length of the longest word in the cell and a line width $c.linewidth$ which is the length of the cell contents when laid out in a single line. The minimum width $C.minwidth$ for a column C is just the maximum of the minimum widths of the cells in the column while the line width $C.linewidth$ is the maximum of the line widths of the cells in the column.

The layout algorithm for standard tables is quite complex [6] but essentially works as follows. Fixed width columns are given the size specified by the user. The widths of the unfixed width columns are scaled to fit the remaining width. The scaling is in terms of the column's line width which is regarded as the column's desired width. However no column is allowed to have width scaled below its minimum width, and column widths are fixed so that they satisfy the ratio width constraints. Once the column widths are computed the cell contents are placed in the cells and the minimum height of each cell computed. The row height is set to to the maximum of the row's cell heights.

Typically the widths are computed using an iterative process. As we have seen, this can give quite bad layout if the cells contain paragraphs of text or multi-column cells.

In our opinion the constraints allowed on table cell layout are ad hoc and unnecessarily restrictive. Here we extend the standard table specification by allowing:

Non-rectangular compound cells. The standard table model allows the designer to specify cells which span more than one column and/or row. However such compound cells are always rectangular. We allow compound cells to consist of arbitrary collections of primitive cells, allowing for example L-shaped regions and even disjoint regions.

To specify this we have added a new (optional) attribute `cluster` which which allows the designer to specify the name of the cluster that the cell belongs to. Cells which have the same value for their cluster attribute are combined to form a single compound cell.

Text flow through arbitrary table cells. The standard table model allows text to flow through only a single cell. We allow text to flow arbitrary collection of possibly compound cells. This allows, for instance, multi-column layout inside tables.

This is specified using a new tag `flow` within a table which is set to a list of cluster names. Text will flow consecutively through the clusters.

Arbitrary linear equality and inequality constraints between column widths and row heights.

Linear equality constraints on column widths generalises the ratio constraint. Thus for instance you can specify that the width of column 1 is equal to the sum of the widths of two other columns or that the width of columns 1, 3 and 5 are equal and the widths of columns 2, 4 and 6 are equal or that the width of column 2 is 1 inch wider than the width of column 3. This is impossible to do with the ratio constraint.

Often the designer would like to state preferences for values. We therefore allow the designer to specify that a constraint inequality or equality is preferred but not required, so that the constraint should be satisfied when possible but does not need to be. This is particularly useful with inequalities since it allows us to specify for instance that a column should be no wider than 20% of the page width but that we would prefer it to be equal to its natural width.

Constraints can refer to column and cell minimum width and line width as well as a convenience measure called the *natural width*. The natural width of a cell is simply defined as 20em while the natural width of a column is the maximum of the natural widths of the cells in the column. We added this for two reasons — firstly it will hopefully encourage page designers to work in relative units more often, and secondly it could be provided as preference for the user, allowing them to choose a comfortable width for easy reading.

We allow constraints to refer to global variables and in particular this allows communication between the constraints for different tables. Thus a designer can specify that the corresponding columns in two different tables have the same width.

We allow required and preferred linear equations and inequalities on row heights as well, and even allow a single constraint to refer to both width and height. This allows us to specify for instance that a particular cell has a fixed aspect ratio or that we wish to minimise the height of the table.

In order to express these new constraints we have extended the XHTML syntax for tables by

- Adding a new tag for linear constraints, the `constraint` tag. This has the attributes `weight`, which, if specified, sets the constraint weight; `id`; and the actual element content is the constraint itself.

- Adding a new tag for named constraint variables. A designer may wish to provide named variables to represent specific concepts in a layout, such as global margin width, border 'units' or even name something purely for documentation purposes. This tag, `var`, has three attributes, `name` which defines the name in the surrounding namespace, `weight`, defining the weight applied to this variable and `goal`, defining the value that this variable aims for. This implicitly adds the constraint $name =_{weight} goal$.

We have added a new namespace to the document, namely the *variable* namespace. The variable name space is used to link variables in the constraint solver to the attributes of the layout. In particular we have variables:

- Defined globally within the document through the `var` tag.
- For the `table` element: $height$ which is the table height, $width$ which is the table width, $col1, col2, \ldots$ and $row1, row2, \ldots$ where $coli$ is the width of column i and $rowj$ the height of row j.
- Each cell has a width and height variable which can be accessed through the attributes, or referenced by `id.width` and `id.height`.

This new namespace is used to build constraints. The constraints are specified where ever a length measure was previously defined. For example, the width of a column may be weakly constrained to be twice the natural width of the first cell's textual contents. We express this by setting the width of the cell like this:

```
<td width="{weak}=2*natural_width">...
```

We use a simple notation for representing non-rectangular cell groups, namely to list the coordinate pairs of all the cells to be grouped. This will need further investigation to determine what is most effective for designers and for automatically generated content.

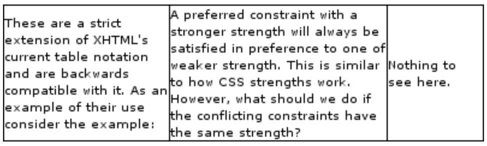
To understand how one might extend table syntax to allow non-rectangular cells, consider the simple table in figure 1. This example might come from a brochure where using up the extra space with photographs would be quite valuable. This can be represented by joining four cells together analogously to the existing 'colspan' / 'rowspan' attributes:

```
<var name="X" goal="290mm" weight="1"/>
<table width="210mm" height="290mm">
 <tr>
  <td cellspan="(0,0),(0,1),(1,1),(0,2)">...</td>
  <td width="=1*X" height="=1*X"><img.../></td>
 </tr>
 <tr></tr>
 <tr>
  <td width="=1*X" height="=1*X"><img.../></td>
 </tr>
</table>
```

These are a strict extension of XHTML's current table notation and are backwards compatible with it. As an example of their use consider the example

```
<table>
 <tr>
  <td width="1*">These are a strict extension of
XHTML's current table notation and are backwards
compatible with it. As an example of their use
consider the example:</td>
  <td width="2*">A preferred constraint with a
stronger strength will always be satisfied in
preference to one of weaker strength. This is
similar to how CSS strengths work. However, what
should we do if the conflicting constraints have
the same strength?</td>
  <td width="20%">Nothing to see here.</td>
 </tr>
</table>
```

and rendered using Mozilla:



These are a strict extension of XHTML's current table notation and are backwards compatible with it. As an example of their use consider the example:	A preferred constraint with a stronger strength will always be satisfied in preference to one of weaker strength. This is similar to how CSS strengths work. However, what should we do if the conflicting constraints have the same strength?	Nothing to see here.

One important issue is how we resolve conflicting preference constraints. Constraint hierarchies [3] formalize such preferences. A constraint hierarchy consists of collections of constraints each labelled with a strength. There is a distinguished strength label *required*: such constraints must be satisfied. (Actually, we will usually omit the "required" label: constraints without a label are assumed to be required.) The other strength labels denote preferences. There can be an arbitrary number of such strengths, and constraints with stronger strength labels are satisfied in preference to ones with weaker strength labels. We will use *weak*, *medium*, *strong* and *very strong* to label non-required constraints and represent them by placing the initial letter next to the appropriate relation (so a weak $\leq$ becomes $\leq_w$).

A preferred constraint with a stronger strength will always be satisfied in preference to one of weaker strength. This is similar to how CSS strengths work. However, what should we do if the conflicting constraints have the same strength? Imagine that we have the constraints

$$\begin{array}{ll} (1) & col1 =_s col2 \\ (2) & col1 =_m 1cm \\ (3) & col2 =_m 3cm \end{array}$$

In this case it would seem reasonable for both $col1$ and $col2$ to be set to 2 cm. This suggest that we should associate a quadratic error penalty with each preferred constraint and find the solution which minimises the sum of these quadratic error penalties. Thus the above is rewritten into

$$\begin{array}{ll} (1') & col1 - col2 = E_1^+ - E_1^- \\ (2') & col1 + E_2^+ - E_2^- = 1 \\ (3') & col2 + E_3^+ - E_3^- = 3 \end{array}$$

where $E_i^\pm$ are hidden variables, and the solution is found by minimising the objective function

$$w_{str} \times (E_1^+ - E_1^-)^2 + w_{med} \times (E_2^+ - E_2^-)^2 + w_{med} \times (E_3^+ - E_3^-)^2$$

Note that our model strictly extends the standard table model. Thus for instance standard layout for our example table is captured by the constraints

$$\begin{array}{llll} (0) & TW & = & col1 + col2 + col3 \\ (1) & TW & \leq_s & page.width \\ (2) & col1 & \geq_v & col1.minwidth \\ (3) & C2 & \geq_v & col2.minwidth \\ (4) & C1 & =_m & S \times col1.linewidth \\ (5) & C2 & =_m & S \times col2.linewidth \\ (6) & C1 & =_s & R \\ (7) & C2 & =_s & 2 \times R \\ (8) & S & \geq & 0 \\ (9) & S & =_w & 1 \\ (10) & C3 & = & 0.2 \times page.width \end{array}$$

where TW is the table width, Ci the width of column i, S a variable capturing the scaling factor and R the variable corresponding to the width $\star$. Constraint (0) captures the table structure, (1) that we really want the table to fit inside the page width, (2) and (3) that unfixed width columns should be wider than their minimum width,

(4) and (5) that we should scale proportionally to a column's line width, (6) and (7) the ratio constraint, (8) that columns have non-negative width, (9) that the line width is the desired width for a column, and (10) that column 3 has a fixed width.

The standard table model tries to make the columns wide enough to display the contents on one line but if this is not possible then uses the relative size of the column's line width to appropriately scale the width of the columns. This usually leads to very compact layout since at least for single paragraph text the line width is in almost exact proportion to the area required by the cell's content and so the model scales column widths proportionally to the area of the maximum sized cell in the column. It does not work so well if there are large multi-column cells or multi-paragraph or non-textual content in some of the cells.

A significant disadvantage of the standard table model is that this criteria for good layout is implicit in the layout algorithm and cannot be modified by the designer unless they override it by giving absolute or relative widths for the column sizes. Our model allows the designer to specify quite complex global optimisation criteria. For example we can specify that we want the layout of our example table to facilitate readability even at the expense of compactness by removing constraints (8) and (9) and replacing (4) and (5) with

$$\begin{array}{lll} (4) & C1 & =_m & \mathit{col1.naturalwidth} \\ (5) & C2 & =_m & \mathit{col2.naturalwidth} \end{array}$$

which will now try to make columns their natural width.

Or if we really do want to ensure compact layout then we can use the constraints

$$\begin{array}{llll} (0) & TW & = & C1 + C2 + C3 \\ (1) & TW & \leq_v & pagewidth \\ (2) & C1 & \geq_v & column1.minwidth \\ (3) & C2 & \geq_v & column2.minwidth \\ (6) & C1 & =_m & R \\ (7) & C2 & =_s & 2 \times R \\ (8) & TH & =_s & 0 \\ (10) & C3 & =_s & 0.2 \times page.width \end{array}$$

which will squash the height until the table hits the page width.

The constraints in our model fall into three main classes:

1. *Structural constraints:* These follow from the nature of tables and are required. They are:
   (1) the width of a table is equal to the sum of the widths of the columns;
   (2) the height of a table is equal to the sum of the heights of the rows;
   (3) a cell is large enough to contain its contents.

2. *General table style constraints:* These capture what style of layout the designer wishes for the table. For instance that layout should be as compact as possible, i.e. that table height be minimised, or that it should be as readable as possible, i.e. that column widths should be close to their natural width.

3. *Table specific constraints:* These capture constraints on cell arrangement, columns and rows that are specific to that table.

Apart from the constraint that a cell is large enough to contain its contents, all of the constraints are linear equality and inequalities which can either be required or be preferred with an associated strength.

One of the key directions CSS3 is taking is multiple column layout. With only a minor extension to the ideas so far presented we can get two new capabilities, namely allowing non-rectangular cells and providing for arbitrary text flow through a table's cells. Multiple column layouts could be implemented through these capabilities.

3. ALGORITHMS FOR TABLE LAYOUT

In this section we investigate three different approaches to solving these constraints. We shall first ignore the problem of handling non-rectangular complex cells and text flow through multiple cells.

3.1 Quadratic programming

As we have discussed, our model of table layout is essentially a quadratic programming problem (apart from the constraint that cells are large enough to contain their content): that is to say we must optimise a convex quadratic objective function subject to linear arithmetic constraints. It is how the cell containment constraint is handled which distinguishes the three approaches.

All our approaches rely on having an incremental solver to solve quadratic programming problems. The solver has a current system of linear arithmetic constraints and a convex quadratic objective function. We assume that it supports the following methods:

- A Boolean function *AddConstraint* which takes a required linear arithmetic constraint and tries to add it to the current system of constraints. When adding the constraint, the solver checks that the new constraint is consistent with the current system. If it is not consistent the constraint is not added and the function returns $false$ otherwise the constraint is added and the function returns $true$.

- A function *AddPrefConstraint* which takes a preferred linear arithmetic constraint and a strength and adds it to the current system of constraints. Note that a preferred constraint can never cause inconsistency so there is no need to check for this. This method adds explicit error variable(s) to the constraint and adds this to the system of constraints and adds the square of these error variables to the objective function.

- A function *RemoveConstraint* which removes a previously added required or preferred constraints from the solver.

- A method *Solve* which solves the quadratic programming problem using the current system of constraints and objective function.

- A method *Val* which takes a variable and returns the value found for that variable

- A method *Soln* which returns the solution θ found in the last call to *Solve*. We assume that θ is a function from variables to values. Thus $\theta(v)$ returns the value of variable v.

- A method *ObjectiveValue* which returns the value of the objective function for the solution found in the last call to *Solve*.

In our implementation we use the Linear Complementary solver from the C++ QOCA toolkit [10] but other quadratic programming packages could be used. It is important for efficiency that the solver is incremental in the sense that when constraints are added or removed it does not solve the problem from scratch but rather makes use of the current solved form.

3.2 Simple two-phase approach

The simplest approach is to first solve the constraints but ignore the cell containment constraints to determine the column widths, then use these widths to determine the minimum height for each

```
two-phase-layout()
add all required and preferred linear constraints to the solver
call Solve
$\theta$ := Soln
for each cell $c_{i,j}$ do
    AddConstraint ($row_j \geq MinHeightCell(c_{i,j}, \theta(col_i))$)
end for
call Solve
$\theta$ := Soln
return $\theta$
```

Figure 4: Two-Phase Algorithm for table layout

cell, and then re-solve the constraints to determine the row heights. The algorithm is given in Figure 4. For simplicity it assumes cells are simple, i.e take up a single row and column. Thus we use the notation $cell_{i,j}$ to refer to the cell in row i column j. We assume that there is a function *MinHeightCell* which takes the name of a cell and the width it is to be laid out in and computes the minimum height that the cell needs to be to contain its contents.

In more detail the algorithm computes the column widths in a first phase which ignores the containment constraint (since it is non-linear). It solves the row heights in second call to the solver, after adding the required constraints that each column has the width computed in the first phase and that each row is high enough to contain all cells in that row laid out using the computed column widths. It returns θ the assignment to row heights and column widths in the final layout.

Since this approach allows arbitrary required and preferred constraints over column width it is more expressive than the standard table layout model. The first phase of the algorithm is similar to that sketched in [2] except that it uses a quadratic penalty for preferred constraints rather than a linear penalty. Providing a second phase means that the author can dictate arbitrary linear constraints on row heights as well as on column heights, specifying for instance that two row heights must be equal and these will be satisfied. However, ignoring the containment constraint in the first phase can lead to bad layout and unnecessarily large cells.

As an example consider

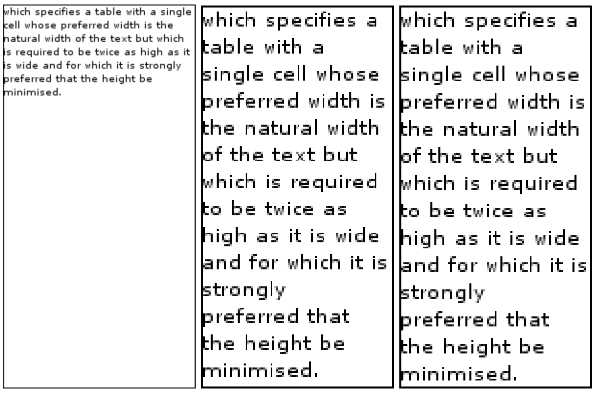
```
<table><tr>
  <td width="{weak}=1.0*natural.width"
      height="=2*width,{strong}=0">...</td>
  </tr></table>
```

which specifies a table with a single cell whose preferred width is the natural width of the text but which is required to be twice as high as it is wide and for which it is strongly preferred that the height be minimised. Figure 5 shows the result of the two pass algorithm on the left.

3.3 Approximating containment constraints

The problem with the Two-Phase Algorithm is that it ignores the containment constraint when computing the cell widths. The problem is that we cannot directly add containment constraints to the quadratic solver since this can only handle linear constraints, whereas text containment constraints are non-linear.

To better understand their characteristics consider Figure 6 which shows the minimal area required by a cell containing a paragraph from this paper as a function of cell width. This is of course discontinuous, corresponding to jumps between the number of lines of text but it is still reasonably constant. Thus a safe, reasonably good lower bound for the area of the cell is the area of the text in the cell ignoring inter-word spacing which we call the cell's *minimal area.*



which specifies a table with a single cell whose preferred width is the natural width of the text but which is required to be twice as high as it is wide and for which it is strongly preferred that the height be minimised.

which specifies a table with a single cell whose preferred width is the natural width of the text but which is required to be twice as high as it is wide and for which it is strongly preferred that the height be minimised.

which specifies a table with a single cell whose preferred width is the natural width of the text but which is required to be twice as high as it is wide and for which it is strongly preferred that the height be minimised.

Figure 5: The two pass, area and branch and bound solution to 2:1 aspect ratio

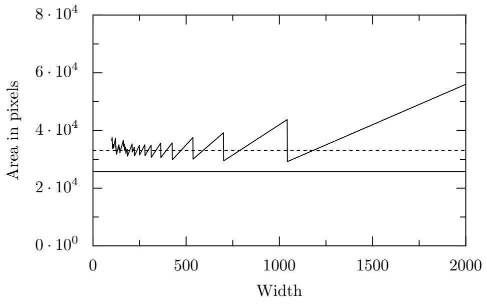


Figure 6: Minimal area of cell versus cell height for example text. The lower constant line is the area of only the words, the upper line is the total area drawing each paragraph on a single line.

```
linear-approx-layout()
add all required and preferred linear constraints to the solver
call Solve
$\theta$ := Soln
repeat
    for each $cell_{i,j}$ do
        $current\_area$ := $\theta(col_i) \times MinHeightCell(cell_{i,j}, \theta(col_i))$
        LinearApproximate($col_i, row_j, current\_area, \theta(col_i), \theta(row_j)$)
    end for
    call Solve
    $\theta$ := Soln
until $\theta$ satisfies containment constraints
remove all linear approximation constraints
for each column $i$ do
AddConstraint($col_i = \theta(col_i)$)
    for each row $j$ do
        AddConstraint ($row_j \geq MinHeightCell(cell_{i,j}, \theta(col_i))$)
    end for
end for
call Solve
$\theta$ := Soln
return $\theta$
```

Figure 7: Linear Approximation Algorithm for table layout

This suggests that we approximate the containment constraint for $cell_{i,j}$ by

$$width(cell_{i,j}) \times height(cell_{i,j}) \geq min\_area(cell_{i,j}). \qquad (1)$$

This has the advantage that it is a continuous constraint, however it is still non-linear and so cannot be directly handled by the quadratic solver.

Fortunately, however it is a convex constraint and so it is possible to model it using a sequence of linear approximations. This is an example of Dynamic Linear Approximation [8] in which non-linear constraints are approximated by linear constraints and is a reasonably well known approach from operations research. The process is best explained using Figure 6 which shows height versus width for a cell and the curve representing the constraint. At any point in the approximation we have a current (w_o, h_o) value for the width and height of the cell. If this does not satisfy Equation 1 then we find the closest point (w_1, h_1) on the curve and then add the linearization of the constraint Equation 1 at (w_1, h_1) to the constraint solver and re-solve.

The precise algorithm is given in Figure 7. It repeatedly adds constraints approximating the current area. Note that is updated rather than being the minimal area. Strictly speaking this may not be a safe approximation but in practice we have found that it is and that it improves the quality of the solution. the only other point to note is that once we have found the solution using the linear approximation we set the column widths to this solution and then remove the linear approximation constraints and recompute the row heights similarly to the second phase of the Two-Phase Algorithm. The reason for doing so is that the approximation may give rise to a row height which is a non-integral number of lines reflecting the conservative nature of the area approximation.

To construct a linear approximation we need to find a point on the constraint, and the associated normal (see Figure 8). We only need to build this constraint if the overall minimum area constraint is not initially satisfied. It is generally best to find the closest point to the goal value that lies on the constant area surface. We do this by solving the minimisation problem:

$$\text{minimise} \quad (x - x_r)^2 + (y - y_r)^2 \quad \text{subject to} \quad xy = 1$$

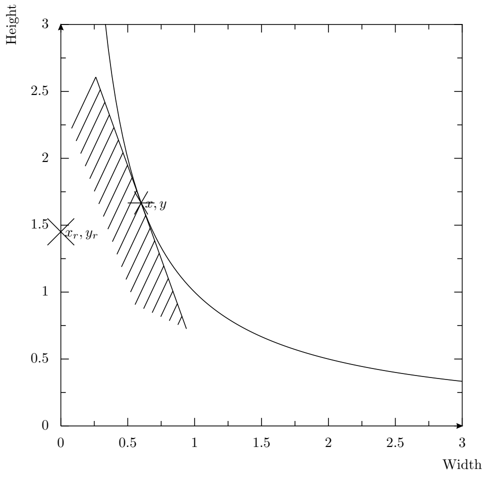


Figure 8: Visualising the linearisation.

where x_r and y_r are the solutions offered by the linear relaxation[2]. This can be solved directly through a standard non-linear goal finder such as Newton/Conjugate Gradient method, but by a simple transformation we can reduce the problem dramatically. Let

$$x = u + v, \qquad y = u - v$$

Then we get

$$\text{minimise} \quad f(X) = (u - u_r)^2 + (u - u_r)^2$$

$$\text{subject to} \quad g(X) \; = \; u^2 - v^2 \; = \; 1$$

We let $\nabla f = \lambda \nabla g$, finding

$$2(u - u_r) = \lambda 2u, \quad 2(v - v_r) = -\lambda 2v, \quad u^2 - v^2 = 1$$

giving the quartic

$$\begin{array}{l} \lambda^4 + \lambda^2(B - A - 2) - 2\lambda(B + A) + 1 + B - A = 0 \\ A = u_r, \qquad B = v_r \end{array} \qquad (\star)$$

(where $A = u_r^2$ and $B = v_r^2$) which we can solve efficiently using a standard polynomial root finder. We take the resulting solution S and generate a half plane using the relation $\nabla g(X) \cdot X \geq \nabla g(X) \cdot S$.

As an example of the algorithm's operation consider the simple one-cell table from above. The table has no given values, so the only constraint on its content is provided by the narrowest width, say 50px, and the line height, say 20px, of the text. The linear constraint solver then finds the solution (50px,25px). The minimal area of the paragraph is 10000, so we construct a half plane at (100px,100px). The linear solver then adjusts the solution according to the designer's preferences, and another half plane added. The process repeats until the error is acceptably minimised.

3.4 Branch-and-bound

However, despite the Linear approximation Algorithm working well on the above example, it is not guaranteed to find the optimal layout. The problem is in the assumption that containment can be approximated by the continuous constraint given in Equation 1. Unfortunately, as we saw in Figure 6 the containment constraint is non-continuous with discontinuities arising when the number of

[2] We can reduce the arbitrary area case by scaling by $1/\sqrt{area}$

lines of text in the cell changes. Thus each cell has a fixed number of different layouts called *configurations* which correspond to different number of lines in the layout.

Our third approach to handling the containment constraints is to model explicitly, and explore, these different configurations. We could use an exhaustive search exploring all combination of cell configurations but this is clearly impractical. Instead we use a variant of the branch-and-bound algorithm [9]. Unlike the previous two algorithms this approach is guaranteed to find the optimal table layout.

Branch-and-bound is typically used to solve *integer programming problems*, that is, linear programming problems in which the variables are required to be integers. It works as follows. First linear programming is used to find the optimal solution of the continuous version of the problem in which all variables may be reals. If all variables in the solution are assigned integers then the solution has been found. Otherwise, there is a variable x whose value d in the solution is not an integer. Two new problems are generated, one with the additional constraint floor$(d) \geq x$ and one with the additional constraint ceil$(d) \leq x$. Each of these is treated in turn using the method above. Eventually a solution with only integer values will be discovered. (Assuming the problem is bounded and satisfiable). This provides the current best solution which is a lower bound on the optimal value. The current best solution is threaded through the computation. If for any sub-problem the current best solution is better than the best solution over the real numbers then that sub-problem can be discarded since it can never yield a better solution over the integers. In the worst case branch-and-bound may have exponential complexity but in practice if the continuous problem is a good approximation to the discrete problem only a small part of the search space needs to be explored.

It is relatively straightforward to give a branch-and-bound algorithm for solving the table layout problem. It is given in Figure 9. The idea is that we use the continuous version of the problem to guide the search. We keep track of which cells currently have a fixed configuration in the array *fixed*. At each step we choose a cell $cell_{i,j}$ which does not yet have a fixed configuration. Given the width w computed for the cell in the continuous approximation we determine the actual height h for the cell required for layout with this width and using the function *ValidForWidths* the minimum w_{min} and maximum w_{max} width for which this configuration with height h is valid. That is, if the width is greater than w_{max} then the cell can be laid out with fewer lines and if it is less than w_{min} more lines are needed. We now generate and explore three new sub-problems. The first is that we commit to this configuration and add the constraints that $w_{min} \leq col_j \leq w_{max}$ and that $row_i \geq h$. The second and third sub-problems are that we do not fix the configuration but instead respectively explore column widths that are narrower than w_{min}, or wider than w_{max}. When we reach a sub-problem for which all cells have a fixed configuration we update the best solution and associated penalty found so far. At each step we call *Solve_Continuous_Approx* which resolves the constraints using the approach of linear approximation. Note that for correctness we must use the conservative lower bound for the cell area. This will return false if the constraints are unsatisfiable or the solution found is worse than the current best solution, in which case exploration of that sub-problem will be terminated.

As branch and bound relies on using the linear system as a guide to the quality, branch and bound does not work well with penalties that aren't monotonic in width. This means that we can't easily add measures like line breaking 'badness' or penalise 'rivers' of whitespace. We have to rely on the line breaking algorithm to try and minimise these artifacts for our requested width.

```
branch-and-bound-layout()
for each $cell_{i,j}$ do
    $fixed[i,j] := false$
end for
$minimum\_so\_far := \infty$
add all required and preferred linear constraints to the solver
if not Solve_Continuous_Approx return $error$
Branch&Bound
return $best\_soln\_so\_far$


Branch&Bound
$\theta :=$ Soln
if for all $i, j$ $fixed[i,j] = true$ then
    if ObjectiveValue $< minimum\_so\_far$ then
        $best\_soln\_so\_far := \theta$
        $minimum\_so\_far :=$ ObjectiveValue
    end if
else
    choose $i, j$ such that $fixed[i,j] = false$
    /* fix the configuration for this cell */
    $h :=$ MinHeightCell$(cell_{i,j}, \theta(col_i))$
    $(w_{min}, w_{max}) :=$ ValidForWidths$(cell_{i,j}, h)$
    /* add constraints to fix configuration checking for satisfiability*/
    if AddConstraints$(w_{min} \leq col_i, w_{max} \geq col_i, h \geq row_j)$ then
        $fixed[i,j] := true$
        if Solve_Continuous_Approx then
            Branch&Bound
        end if
        RemoveConstraints$(h \geq row_j, w_{max} \geq col_i, w_{min} \leq col_i)$
    end if
    /* now explore narrower configurations for this cell */
    if AddConstraint$(w_{min} > col_i)$ then
        if Solve_Continuous_Approx then
            Branch&Bound
        end if
        RemoveConstraint$(w_{min} > col_i)$
    end if
    /* now explore wider configurations for this cell */
    if AddConstraint$(w_{max} < col_i)$ then
        if Solve_Continuous_Approx then
            Branch&Bound
        end if
        RemoveConstraint$(w_{max} < col_i)$
    end if
end if
```

Figure 9: Branch-and-bound algorithm for solving table layout

Branch and bound seems to give the most gain when there is a big difference between solutions. This occurs mostly for cells with only a small number words, say, less than 10. A promising avenue for further research is only branching where the configurations are more 'lumpy' than a certain threshold, potentially gaining the improvement given by branch and bound whilst only incurring a slight performance penalty.

How do we determine a layout for non-rectangular regions such as the brochure discussed earlier? We start as before by filling out all the constraints elsewhere in the system. We again find a linear approximation to the area constraint:

$$\text{minimise} \quad \sum_i (x_i - x_{ri})^2 + (y_i - y_{ri})^2 \quad \text{subject to} \quad \sum_i x_i y_i = 1$$

that is, the total area of the cell group must be greater than 1. Working through the mathematics we find that we only need to solve the quartic given in ($\star$), this time letting $A = \sum_i u_{ri}^2$ and $B = \sum_i u_{ri}^2$.

Example	Two pass	Area	Branch & bound
simple(linear)	40ms	370ms	660ms
simple (natural)	30ms	600ms	890ms
simple (compact)	40ms	270ms	390ms
two and two	60ms	90ms	300ms
two and two (2:1 ratio)	60ms	150ms	2000ms
brochure	60ms	290ms	770ms

Figure 10: The times taken to lay out our examples.

Example	Rows and Columns	Two pass		Area		B-and-b	
		v	c	v	c	v	c
simple (linear)	1, 3	15	17	15	65	15	80
simple (natural)	1, 3	11	15	11	66	11	89
simple (compact)	1, 3	9	14	9	92	9	97
two and two	3, 3	13	21	13	45	13	47
2 and 2 (2:1 ratio)	3, 3	16	24	16	54	16	77
brochure	3, 2	14	20	14	64	14	103

Figure 11: The number of variables and constraints required for our examples.

This time we can't specify a minimum height our width for the cells in our cell group. This turns out not to be a problem. If we start with all the cells in the group having zero size and otherwise unconstrained, then the half plane will constrain all the cells to be square, each getting a even fraction of the total area. At this point, our normal continuous updating approximation will work and provide a good initial solution.

If we wish to further refine our solution then with a bit of modification we can use the branch and bound refinement as per algorithm three. Rather than just refining a single height and width, however, we now must choose a configuration for each sub-cell. It is also useful to work through the cells in the order in which they are filled. We have found it helpful to group cells horizontally to reduce the amount of searching required.

Clearly there is nothing special about the order in which we draw the text from the point of view of the solvers. Thus, we can define an arbitrary order for the text through the table's cells. This allows text to flow through multiple cells.

4. EVALUATION

We now evaluate the effectiveness of the three approaches we have introduced in the last section. All tests were performed on a 400MHz PowerPC system using Python 2.3 and Qoca for the implementation. Only the time spent doing the layout problem are given. Profile indicates that a large portion of time is spent determining the text extents of a particular layout choice, rewriting this in C would probably improve the numbers significantly.

Table 11 provides some statistics about each example, giving the number of rows, columns, the number of variables in the system and the total number of linear constraints.

5. CONCLUSION

Tables are one of the most powerful and useful design elements in current web document standards. However support for automatic table layout is still limited. We show how viewing table layout as a constrained optimisation problem leads us to naturally generalise the current model. Our generalisation allows the designer to specify required and preferred linear constraints over the column widths and row heights. We also allow cells to be non-rectangular regions and we allow text to flow between arbitrary cells.

We present and evaluate three possible approaches to table layout, all of which are based on quadratic programming techniques. The first is a two-phase approach in which column widths are determined in a first phase without directly taking into account the cell content, while the second phase computes row heights. The second approach is to use quadratic programming in conjunction with a dynamic linear approximation to the cell containment constraint. The third approach is to use branch and bound.

Our results are extremely promising. In particular the second approach is comparable in speed to the current layout algorithms but gives better layout for tables with multi-column and multi-row cells or tables with multiple paragraphs in a cell. This approach also handles much more expressive designer constraints and non-rectangular cells and text flow between arbitrary cells. These last two points are very important for when tables are used for page layout rather than just the display of tabular information.

Our work not only has relevance to table layout. Clearly it forms the basis for better and more expressive page layout. This is something we are now looking at. It also has applications in presentation and printing of spread sheets and for PowerPoint-like presentation software.

Something else we have also considered is support for alternate content. This is exactly the sort of problem that branch-and-bound was developed to solve. Our initial experimentation is extremely promising. It is straightforward to extend the Branch-and-Bound Algorithm to handle alternatives. We just branch on each alternative, adding the appropriate penalty when we commit to an alternative. However a better approach is to only use Branch-and-Bound to explore alternatives and to use the Linear Approximation Algorithm to explore different text configurations.

Acknowledgements

We would like to thank Peter Moulder for spending time improving Qoca for our needs, and Prof. Richard Anderson for the precise explanation of his earlier work and further thoughts.

6. REFERENCES

[1] R. J. Anderson and S. Sobti. The table layout problem. In *COMPGEOM: Annual ACM Symposium on Computational Geometry*, pages 115–123, 1999.

[2] G. J. Badros, A. Borning, K. Marriott, and P. Stuckey. Constraint cascading style sheets for the web. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*, pages 73–82, New York, Nov. 1999. ACM.

[3] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, Sept. 1992.

[4] A. Borning, R. Lin, and K. Marriott. Constraints for the web. In *Proceedings of ACM MULTIMEDIA'97*, pages 173–182, Nov. 1997.

[5] A. Borning, R. Lin, and K. Marriott. Constraint-based document layout for the web. *Multimedia Systems*, 8(3):177–189, 2000.

[6] B. Bos, H. W. Lie, C. Lilley, and I. Jacobs. Cascading style sheets, level 2, tables section. W3C Working Draft, Jan. 1998. http://www.w3.org/TR/CSS2/tables.html.

[7] R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, Chichester, New York, Brisbane, Toronto, Singapore, 1987.

[8] N. Hurst, K. Marriott, and P. Moulder. Dynamic approximation of complex graphical constraints by linear constraints. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 191–200. ACM Press, 2002.

[9] E. Lawler and D. Wood. Branch-and-bound methods: a survey. pages 699–719, 1966.

[10] K. Marriott, S. Chok, and A. Finlay. A tableau based constraint solving toolkit for interactive graphical applications. In *International Conference on Principles and Practice of Constraint Programming (CP98)*, pages 340–354, 1998.

[11] X. Wang and D. Wood. Tabular formatting problems. In *3rd Principles of Document Processing*, pages 171–181, 1996.