INFORMS JOURNAL ON COMPUTING
Vol. 00, No. 0, Xxxxx 0000, pp. 000–000
ISSN 0899-1499 | EISSN 1526-5528 | 00 | 0000 | 0001

INFORMS
DOI 10.1287/xxxx.0000.0000

Automatic Minimal-Height Table Layout

Mihai Bilauca
University of Limerick
Limerick, mihai@bilauca.net

Graeme Gange
Dept of CSSE
University of Melbourne
Vic. 3010, Australia, ggange@cs.mu.oz.au

Patrick Healy
University of Limerick
Limerick, patrick.healy@ul.ie

Kim Marriott
Faculty of IT and NICTA Victoria
Monash University
Vic. 3800, Australia, kim.marriott@monash.edu

Peter Moulder
Faculty of IT
Monash University
Vic. 3800, Australia, peter.moulder@monash.edu

Peter Stuckey
Dept of CSSE
University of Melbourne
Vic. 3010, Australia, pjs@cs.mu.oz.au

Automatic layout of tables is useful in word processing applications and is required in on-line applications because of the need to tailor the layout to viewport width, choice of font and dynamic content. However, if the table contains text, minimizing the height of the table for a given maximum width is a difficult combinatorial optimization problem because of the need to find the right choice of height/width configuration for each cell in the table. We investigate the modelling decisions involved in formulating this problem for use with standard combinatorial optimization techniques that are guaranteed to find the minimal height table. To the best of our knowledge we are the first to do so. We provide a detailed empirical evaluation of the resulting models using MIP and constraint programming with lazy clause generation.

Key words: automatic table layout, constrained optimization, typography

1. Introduction

Tables are provided in virtually all document formatting systems and are one of the most powerful and useful design elements in current web document standards such as (X)HTML. For on-line presentation it is not practical to require the author to specify table column widths at document authoring time since the layout must be adjusted to different viewport widths and to different-sized text. For instance, the viewer may choose a larger font. Dynamic content is another reason that it may be impossible for the document author to fix the table column widths. This is an issue for both web pages and for variable data printing (VDP) in which print material is customized to a particular recipient. Good automatic layout of tables is therefore needed for both on-line and VDP applications and is useful in other document processing applications since it reduces the burden on the author of formatting tables.

However, automatic layout of tables that contain text is computationally difficult. The reason is that if a cell contains text then this implicitly constrains the cell to take one of a discrete number of possible width/height configurations arising from different line breaking choices. Because of the need to choose the configuration for each cell, Anderson and Sobti (1999) have shown that table layout with text is NP-hard for reasonable layout requirements such as minimizing table height for a given width.

In this paper we are concerned with complete techniques that are guaranteed to find the optimal solution. While these are necessarily non-polynomial in the worst case (unless P=NP) we are interested in finding out if they are practical for small and medium sized table layout. Furthermore, even if the complete techniques are impractical for normal use, it is still worthwhile to develop complete methods because these provide a benchmark with which to compare the quality of layout of heuristic techniques proposed for web browsers and document processing software. For instance, while Gecko (the layout engine used by the Firefox web browser) provides sophisticated HTML/CSS rendering, Figure 1 shows that its automatic table layouts can be far from the most compact as computed by the algorithms we present.

This is the first paper to look at complete constrained optimization techniques for finding minimal height table layouts. We give a number of different ways of modelling table layout

This paper collates work presented at DocEng'10 (Bilauca and Healy 2010) and DocEng'11 (Bilauca and Healy 2011, Gange et al. 2011). We also present several improved models, and an extended discussion and evaluation of the different modelling decisions.

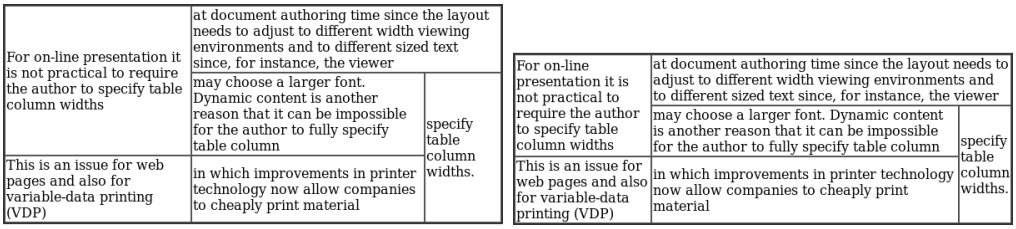

Figure 1 Example table comparing layout using Gecko (on the left) with the minimal height layout (on the right).

and investigate two approaches to solving these models, based on generic approaches for solving combinatorial optimization problems that have proven to be useful in a wide variety of practical applications. The first approach uses a traditional MIP encoding of the models. Our second approach uses constraint programming (CP) (Marriott and Stuckey 1998). We use a state-of-the-art hybrid solving approach, lazy clause generation (Ohrimenko et al. 2009), which combines CP and SAT technology. The advantage of the hybrid approach is that during search it learns *nogoods* that prevent it from repeating similar search later on, and it tracks *activity* of decisions, and uses an automatic search approach that concentrates on decisions likely to lead to early failure. This can potentially drastically reduce the search space, if the reasons for failure are lifted from the nm cell variables to the $n+m$ row/column variables.

We provide an extensive empirical evaluation of these approaches. We first compare the approaches on a large body of tables collected from the web. This comprised more than 2000 tables that were hard to solve in the sense that the standard HTML table layout algorithm did not find the minimal height layout. Most methods performed well on this set of examples and solved almost all problems in less than 1 second. We then tested the scalability of the algorithms on some artificial table layout examples of increasing size. In this case we found that the "cell-free" model was the most robust approach, with both CP and MIP approaches being competitive.

In the next section we review related work. In Section 3, we provide a formal definition of the table layout problem and give a number of ways of modelling it. In Sections 4 and 5 we give models for solving the layout problem using MIP and CP techniques respectively while Section 6 gives the empirical evaluation.

2. Related Work

Our review of table layout research is based on that of the recent review of automatic document formatting by Hurst et al. (2009). Starting with Beach (1985), a number of

authors have investigated automatic table layout from a constrained optimization viewpoint. Beach considered packing rectangular fixed sized cells into a grid and showed this can be done in polynomial time. He did not consider the case where cells could contain text with multiple line breaking choices. Wang and Wood (1997) gave a branch and bound algorithm accelerated with a polynomial-time greedy algorithm for finding a table layout satisfying linear designer constraints on the column widths and row heights. They showed that the associated decision problem was NP-Hard. They modelled table layout as a satisfaction problem rather than an optimization problem. In the latter we are interested in minimising an objective function (the table height) rather than just finding any solution which satisfies the constraints.

Anderson and Sobti (1999) showed that finding the minimum height layout for a fixed maximum width is NP-Complete for simple tables even without designer constraints. They gave two heuristic methods for finding a minimum height layout for simple cells. The first was based on encoding table layout as the problem of finding the minimum cut in a flow graph while the second is a linear programming approximation to the problem in which the convex hull of the configurations is modelled using a conjunction of linear inequality constraints. Using a continuous linear approximation to the constraint that a cell is large enough to contain its content has been suggested by a number of other researchers such as Lin (2006).

Beaumont (2004) and Hurst et al. (2005) suggested a non-linear continuous approximation in which the area of each cell is constrained to be greater than the area of its content (when laid out in a single line). Beaumont used the non-linear solver MINOS to solve the resulting non-linear problem while Hurst et al. noted that it was a convex optimization problem and could be modelled using conic programming and solved using polynomial time interior point methods. Hurst et al. (2006) have given a more efficient specialized variable elimination method for solving a simplified form of the continuous approximation for simple tables.

Hurst et al. (2005) also suggested a polynomial-time heuristic in which the table is laid out by starting from the narrowest possible layout for the table and then iteratively widening a column, choosing the column that leads to the most reduction in height for least increase in width. This heuristic was further explored in Marriott et al. (2013).

Another heuristic approach to table layout is *column-driven layout*. In this approach three widths – an ideal width, and a minimum and maximum width – are computed for each column and the columns are proportionately scaled down/up from their ideal size (but not below their minimum size or above their maximum size) until the table has the desired width. The row heights are then computed by laying out the content of the cells in each row. The standard table layout algorithm suggested for HTML, CSS and XSL (Raggett et al. 1999) is another example of a column-driven approach. Other column-driven layout approaches include Borning et al. (2000), Badros et al. (1999). These allow the designer to specify required and preferred linear arithmetic constraints over column widths and use a linear constraint solver to determine the column widths. Lutteroth and Weber (2006) also allow linear constraints over column widths in their extension of standard table layout which allows columns to be partially ordered rather than totally ordered. None of these approaches directly minimize table height.

Thus we see that, apart from Wang and Word's work in 1985, all previous research into table layout with breakable text has focussed on developing heuristic techniques. Here our focus is on approaches that are guaranteed to find an optimal table layout in the sense that the table height is minimized. This is in contrast to Wang and Wood who were concerned with finding a layout that satisfied the designer constraints, rather than minimizing table height.[1]

A preliminary version of the methods described here have appeared in three earlier conference papers (Bilauca and Healy 2010, 2011, Gange et al. 2011): the current paper extends the conference versions by introducing a unified exposition of the different models and a new encoding (cell-free) which outperforms the earlier models, fixes some errors and provides a systematic empirical evaluation of all the different approaches. [2]

3. Modelling the Table Layout Problem

We assume throughout this paper that the table of interest has n columns and m rows. A *layout* (w, h) for a table is an assignment of widths, w, to the columns and heights, h, to

[1] Though one could minimize table height by repeatedly searching for a feasible solution with a table height less than the best solution so far.

[2] An additional A^*-based approach was presented in Gange et al. (2011). This method was not competititve, and does not support the modelling choices discussed in later sections; it has thus been omitted here.

the rows where w_c is the width of column c and h_r the height of row r. We make use of the width and height functions:

$$wd_{c_1,c_2}(w) = \sum_{c=c_1}^{c_2} w_c, \qquad ht_{r_1,r_2}(h) = \sum_{r=r_1}^{r_2} h_r$$

where $wd_{c_1,c_2}(w)$ gives the sum of the column widths from columns c_1 to c_2 inclusive, and $ht_{r_1,r_2}(h)$ gives the sum of row heights from row r_1 to r_2 inclusive; hence $ht_{1,m}(h)$ and $wd_{1,n}(w)$, give the overall table height and width respectively.

The designer specifies how the grid elements of the table are partitioned into logical elements or *cells*. We call this the *table structure*. A *simple* cell spans a single row and column of the table while a *compound* (or spanning) cell consists of multiple grid elements forming a rectangle, i.e. the grid elements *span* contiguous rows and columns. Compound cells complicate table layout.

If d is a cell we define $rows(d)$ to be the rows in which d occurs and $cols(d)$ to be the set of columns spanned by d. We let

$$bot(d) = \max\ rows(d),\ top(d) = \min\ rows(d),$$
$$left(d) = \min\ cols(d),\quad right(d) = \max\ cols(d).$$

and, letting *Cells* be the set of cells in the table, for each row r and column c we define

$$lcells_c = \{d \in Cells \mid left(d) = c\},$$
$$rcells_c = \{d \in Cells \mid right(d) = c\},$$
$$cells_c = \{d \in Cells \mid c \in cols(d)\},$$
$$bcells_r = \{d \in Cells \mid bottom(d) = r\}.$$

Each cell d has a minimum width, $minw(d)$, which is typically the length of the longest word in the cell, and a minimum height $minh(d)$, which is typically the height of the highest text element in the cell.

The main decision in table layout is how to break the lines of text in each cell. Different choices give rise to different width/height cell configurations. Cells have a number of *minimal configurations* where a minimal configuration is a pair (w, h) such that the text in the cell can be laid out in a rectangle with width w and height h but there is no smaller rectangle for which this is true. That is, for all $w' \leq w$ and $h' \leq h$ either $h = h'$ and $w = w'$, or the text does not fit in a rectangle with width w' and height h'. These minimal configurations

are *anti-monotonic* in the sense that if the width increases then the height will strictly decrease. For text with uniform height with W words (or more exactly, W possible line breaks) there are up to W minimal configurations, each of which has a different number of lines. In the case of non-uniform height text there can be no more than $O(W^2)$ minimal configurations.

A number of algorithms have been developed for computing the minimal configurations of the text in a cell (Hurst et al. 2009). Here we assume that these are pre-computed and that

$$C_d = [(w_1, h_1), ..., (w_{N_d}, h_{N_d})]$$

gives the width/height pairs for the minimal configurations of cell d sorted in increasing order of width. We will make use of the function $minheight(d, w)$ which gives the minimum height $h \geq minh(d)$ that allows the cell contents to fit in a rectangle of width $w \geq minw(d)$. This can be readily computed from the list of configurations.

Designer constraints specify relationships between the column widths and/or row heights. These are specific to a particular table. Useful designer constraints include:

- *fixed size* for selected column widths or rows
- *fixed ratios* between selected column widths or between selected rows

For simplicity we do not consider nested tables or designer constraints until Section 7.

The table *layout style* captures what is required in a good layout. We shall focus on the *minimum height* layout style, i.e. find a layout that minimizes the table height for a particular table width. For simplicity, we assume that the viewport is wide enough to allow the table to be laid out.

We have explored a number of different ways of modelling table layout since, as we shall see, efficiency depends crucially on the choice of model. Our starting point is the model:

$$\begin{aligned}
&\text{find } w \text{ and } h \text{ that minimize } ht_{1,m}(h) \text{ subject to} \\
&\quad (cw_d, ch_d) \in C_d, && \forall d \in Cells && (3.1)\\
\wedge\ & wd_{left(d),right(d)}(w) \geq cw_d, && \forall d \in Cells && (3.2)\\
\wedge\ & ht_{top(d),bot(d)}(h) \geq ch_d, && \forall d \in Cells && (3.3)\\
\wedge\ & wd_{1,n}(w) \leq W && && (3.4)
\end{aligned}$$

Since the decision variables in this model are the row heights h and column widths w we call it the *extent* table layout model.

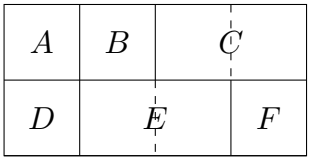

Figure 2 A small instance with column-spans.

EXAMPLE 1. Consider the table shown in Figure 2. The basic extent model is constructed:

$$
\begin{aligned}
&\text{find } p \text{ and } q \text{ that minimize } p_m \text{ subject to}\\
&(cw_d, ch_d) \in C(d),\ \forall\ d \in \{A, \ldots, F\}\\
&\wedge \begin{pmatrix} q_{c_1} \geq cw_A \\ \wedge\ q_{c_2} \geq cw_B \\ \wedge\ q_{c_3} + q_{c_4} \geq cw_C \\ \wedge\ q_{c_2} + q_{c_3} \geq cw_E \\ \wedge\ q_{c_4} \geq cw_F \end{pmatrix} \quad \wedge \begin{pmatrix} \wedge\ p_{r_1} \geq ch_A \\ \wedge\ p_{r_1} \geq ch_B \\ \wedge\ p_{r_1} \geq ch_C \\ \wedge\ p_{r_2} \geq ch_D \\ \wedge\ p_{r_2} \geq ch_E \\ \wedge\ p_{r_2} \geq ch_F \end{pmatrix}\\
&\wedge\ q_{c_1} + q_{c_2} + q_{c_3} + q_{c_4} \leq W
\end{aligned}
$$

□

An alternate model is to determine the *positions* of the rows p and columns q rather than their widths and heights. The variable p_r gives the bottom of row r and q_c the right hand side of column c. We call this the *positional* model. Note that the variables in the two models are related by $p_r = ht_{1,r}$ and $q_c = wd_{1,c}$ and we define

$$wd_{c_1,c_2}(q) = q_{c_2} - q_{c_1-1}, \qquad ht_{r_1,r_2}(p) = p_{r_2} - p_{r_1-1}$$

The model is:

$$
\begin{aligned}
&\text{find } p \text{ and } q \text{ that minimize } p_m \text{ subject to}\\
&\quad (cw_d, ch_d) \in C_d, && \forall d \in Cells\\
&\wedge\ wd_{left(d),right(d)}(q) \geq cw_d, && \forall d \in Cells\\
&\wedge\ ht_{top(d),bot(d)}(p) \geq ch_d, && \forall d \in Cells\\
&\wedge\ q_n \leq W\\
&\wedge\ q_0 = 0\ \wedge\ p_0 = 0\\
&\wedge\ q_c \leq q_{c+1}, && c = 1, \ldots, n-1\\
&\wedge\ p_r \leq p_{r+1}, && r = 1, \ldots, m-1
\end{aligned}
$$

This has the potential advantage that the width/height constraints associated with compound cells contain fewer variables.

An unfortunate property of the positional model is that if q_c changes, the values of $[q_{c+1}, \ldots, q_n]$ will all also change (similarly for p_r). In a solver where variable bounds are maintained, this can cause considerable degradation. To reduce this impact, we can decompose the columns (and rows) into contiguous *blocks*, such that every column (resp. row) span is contained strictly within a single block. We then encode the width of each column-block using the positional model, and require that the sum of block widths is no greater than w. We call this the *position-block* model. In the worst case, of course, we may be unable to decompose the columns, and we end up with the positional encoding. In practice, however, the blocks tend to be quite small.

Example 2. Consider again the table illustrated in Figure 2. While there are cells spanning across the end of columns 2 and 3, there are no cells crossing the end of column 1. As such, we can partition the columns into regions $\{[1,2),[2,5)\}$, encoding the width of each region using the positional encoding, and combining the regions using the extent encoding. This yields the following modified width constraints:

$$\begin{aligned} & wd_{1,2}(w) \geq cw_d,\ \forall d \in \{A, D\} \\ \wedge\ & wd_{2,3}(w) \geq cw_B \\ \wedge\ & wd_{2,4}(w) \geq cw_E \\ \wedge\ & wd_{2,5}(w) \geq wd_{2,3}(w) + cw_C \\ \wedge\ & wd_{2,5}(w) \geq wd_{2,4}(w) + cw_F \\ \wedge\ & wd_{1,2}(w) + wd_{2,5}(w) \leq W \end{aligned}$$

□

One possible way of improving the model is to restrict the choice of column and row positions or widths based on the configurations of the cells in the row or column. In the case of a simple table the row heights or column widths can be restricted to being the height or width of one of the configurations of the cells in that row or column. For column c and row r we define

$$\begin{aligned} \mathit{configs}_c &= \mathit{sort}_<([w \mid \exists d \in \mathit{Cells},\ \exists h,\ c = \mathit{left}(d) = \mathit{right}(d) \wedge (w,h) \in C_d]) \\ \mathit{configs}_r &= \mathit{sort}_<([h \mid \exists d \in \mathit{Cells},\ \exists w,\ r = \mathit{top}(d) = \mathit{bot}(d) \wedge (w,h) \in C_d]) \end{aligned}$$

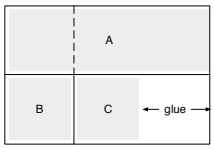

Figure 3 **Example table showing that if there are column spans then the column widths cannot be restricted to column width configurations**

where $sort_<$ returns a list of values in ascending order. We can then add the redundant constraints to the model

$$\begin{aligned} & w_c \in \mathit{configs}_c,\ c = 1, \ldots, n \\ \wedge\ & h_r \in \mathit{configs}_r,\ r = 1, \ldots, m. \end{aligned}$$

In the case of tables with cell or row spans it is more difficult to restrict the row heights or column widths. Consider the table in Figure 3 in which compound cell A has a single configuration $(3, 1)$ and simple cells B and C have the single configuration $(1, 1)$. Clearly in this case we cannot restrict the width of column 2 to the width configurations of cell C. One way to handle this is to allow extra spacing in the columns and rows that are spanned by a compound cell, we call this spacing "glue" because of its similarity to glue in TEX.

For each column c and row r we introduce a new non-negative variable g_c and f_r. The model becomes

$$\begin{aligned} & \text{find } w,\ g,\ h \text{ and } f \text{ that minimize } \textstyle\sum_{r=1}^{m}(h_r + f_r) \text{ subject to} \\ & \quad (cw_d, ch_d) \in C_d, && \forall d \in \mathit{Cells} \\ \wedge\ & w_c \geq cw_d, && \forall d \in \mathit{Cells} \text{ s.t. } \exists c,\ left(d) = right(d) = c \\ \wedge\ & \textstyle\sum_{c=left(d)}^{right(d)}(w_c + g_c) \geq cw_d, && \forall d \in \mathit{Cells} \text{ s.t. } left(d) \neq right(d) \\ \wedge\ & h_r \geq ch_d, && \forall d \in \mathit{Cells} \text{ s.t. } \exists r,\ top(d) = bot(d) = c \\ \wedge\ & \textstyle\sum_{r=top(d)}^{bot(d)}(h_r + f_r) \geq ch_d, && \forall d \in \mathit{Cells} \text{ s.t. } top(d) \neq bot(d) \\ \wedge\ & \textstyle\sum_{c=1}^{n}(w_c + g_c) \leq W \\ \wedge\ & f_r \geq 0\ \wedge\ h_r \in \mathit{configs}_r, && r = 1, \ldots, m \\ \wedge\ & g_c \geq 0\ \wedge\ w_c \in \mathit{configs}_c, && c = 1, \ldots, n \end{aligned}$$

If a column or row contains no simple cells then it has a dummy 0 width or height configuration. We can further constrain this model by setting g_c and f_r equal to 0 if there are no column spans or row spans that finish on that row or column. We call this the *glue model*.

Example 3. Consider again the width constraints for the table in Example 1, where the set of minimal configurations are given by:

$$C(d) = \begin{cases} \{(1,2),(3,1)\} & \textbf{if } d \in \{A,B,D,F\} \\ \{(1,3),(2,2),(4,1)\} & \textbf{if } d \in \{C,E\} \end{cases}$$

Column c_1 contains only simple cells, so $g_{c_1} = 0$. c_3 contains no simple cells, so q_{c_3} may be fixed to 0. This yields the following encoding.

$$\begin{aligned} & \dots \\ \wedge \quad & q_{c_1} \in \{1,3\} \wedge g_{c_1} = 0 \\ \wedge \quad & q_{c_2} \in \{1,3\} \wedge g_{c_2} \geq 0 \\ \wedge \quad & q_{c_3} = 0 \wedge g_{c_3} \geq 0 \\ \wedge \quad & q_{c_4} \in \{1,3\} \wedge g_{c_1} \geq 0 \\ \wedge \quad & q_{c_1} \geq cw_A \\ \wedge \quad & q_{c_2} \geq cw_B \\ \wedge \quad & (q_{c_3} + q_{c_4}) + (g_{c_3} + g_{c_4}) \geq cw_C \\ \wedge \quad & (q_{c_2} + q_{c_3}) + (g_{c_2} + g_{c_3}) \geq cw_E \\ \wedge \quad & q_{c_4} \geq cw_F \\ \wedge \quad & (q_{c_1} + q_{c_2} + q_{c_3} + q_{c_4}) + (g_{c_1} + g_{c_2} + g_{c_3} + g_{c_4}) \leq W \end{aligned}$$

□

An alternative method for restricting column and row positions when there are column or row spans is to generalise the idea of column and row configurations to row and column spans. We call this the *row and column span value* (RCSV) model. We define

$$\begin{aligned} configs_{c_1,c_2} &= sort([w \mid \exists d \in Cells,\ \exists h,\ c_1 = left(d)\ \wedge\ c_2 = right(d) \wedge (w,h) \in C_d]) \\ configs_{r_1,r_2} &= sort([h \mid \exists d \in Cells,\ \exists w,\ r_1 = top(d)\ \wedge\ r_2 = bot(d) \wedge (w,h) \in C_d]) \\ spans_c &= \{left(d) \mid \exists d \in Cells,\ c = right(d)\} \\ spans_r &= \{top(d) \mid \exists d \in Cells,\ r = bot(d)\}. \end{aligned}$$

We can add the (redundant) constraints to the extent model:

$$\begin{aligned} & h_r \in configs_r \wedge && \forall r \in 1,\dots,m \text{ s.t. } spans(r) = \{r\} \\ \wedge \quad & h_{r,r'} \in configs_{r,r'} \ \wedge\ ht_{r,r'} \geq h_{r,r'} && \forall r \in 1,\dots,m,\ \forall r' \in spans_r \text{ s.t. } spans(r) \neq \{r\} \\ \wedge \quad & w_c \in configs_c && \forall c \in 1,\dots,n,\ \text{s.t. } spans(c) = \{c\} \\ \wedge \quad & w_{c,c'} \in configs_{c,c'} \ \wedge\ wd_{c,c'} \geq w_{c,c'} && \forall c \in 1,\dots,m,\ \forall c' \in spans_c. \text{ s.t. } spans(c) \neq \{c\} \end{aligned}$$

We can add similar constraints to the positional model.

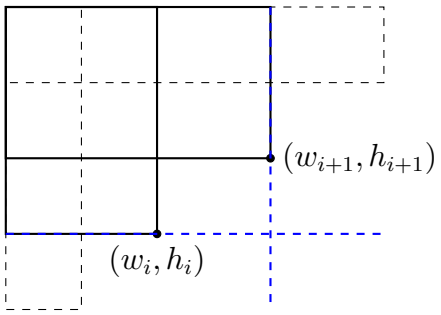

Figure 4 **Two adjacent configurations for a cell. Any feasible configuration must have either at least width w_{i+1}, or height at least h_i.**

EXAMPLE 4. Consider the cell configurations described in Example 3. The spans of interest are $\{[1,2),[2,3),[2,4),[3,5),[4,5)\}$. The span $[1,2)$ is a single column consisting only of simple cells, so can be represented directly with q_{c_1}. We must introduce an additional variable $w_{c,c'}$ for the remaining spans. Applying this to the extent encoding yields the following width constraints:

$$\begin{array}{c} \cdots \\ \wedge \begin{pmatrix} q_{c_1} \in \{1,3\} \\ \wedge \quad w_{c_2,c_3} \in \{1,3\} \\ \wedge \quad w_{c_2,c_4} \in \{1,2,4\} \\ \wedge \quad w_{c_3,c_5} \in \{1,2,4\} \\ \wedge \quad w_{c_4,c_5} \in \{1,3\} \end{pmatrix} \wedge \begin{pmatrix} q_{c_1} \geq cw_A \\ \wedge \quad w_{c_2,c_3} \geq cw_B \\ \wedge \quad w_{c_3,c_5} \geq cw_C \\ \wedge \quad w_{c_2,c_4} \geq cw_E \\ \wedge \quad w_{c_4,c_5} \geq cw_F \end{pmatrix} \wedge \begin{pmatrix} q_{c_2} \geq w_{c_2,c_3} \\ \wedge \quad q_{c_2} + q_{c_3} \geq w_{c_2,c_4} \\ \wedge \quad q_{c_3} + q_{c_4} \geq w_{c_3,c_5} \\ \wedge \quad q_{c_4} \geq w_{c_4,c_5} \end{pmatrix} \\ \wedge \; q_{c_1} + q_{c_2} + q_{c_3} + q_{c_4} \leq W \end{array}$$

This increases the number of variables in the problem, but can reduce the search space if the cell configurations are sparse. □

A crucial decision in the model is how to model the choice of configuration. Until now we have explicitly modelled the choice of configuration (cw_d, ch_d) for cell d using a list membership constraint. However, because the height of the configurations varies anti-monotonically with the width, we can use an alternate model.

Let $C_d = [(w_1,h_1),...,(w_N,h_N)]$ be the configurations for cell d ordered according to increasing width (or, equivalently, decreasing height). Consider two adjacent configurations, (w_i,h_i) and (w_{i+1},h_{i+1}). As Figure 4 illustrates, if the cell is in a configuration $c \leq i$, then $h_c \geq h_i$. If the cell is in a configuration $c \geq i+1$, then $w_c \geq w_{i+1}$. By introducing these

constraints for each pair of adjacent configurations, we obtain an alternate encoding of $(cw_d, ch_d) \in C_d$ by

$$cw_d \geq w_1 \wedge \bigwedge_{i=1}^{N-1} (ch_d \geq h_i \vee cw_d \geq w_{i+1}) \wedge ch_d \geq h_N.$$

We call this the *cell-free encoding*. It can be used with any of the previous models.

EXAMPLE 5. Consider again cell C from Example 3, with minimal configurations $\{(1,3),(2,2),(4,1)\}$. Rather than an explicit membership constraint, we can encode the configurations as follows:

$$\begin{aligned} & wd_C \geq 1 \wedge ht_C \geq 1 \\ \wedge \quad & wd_C \geq 2 \vee ht_C \geq 3 \\ \wedge \quad & wd_C \geq 4 \vee ht_C \geq 2 \\ \wedge \quad & ht_C \geq 1 \end{aligned}$$

□

Observe that the effectiveness of this encoding is dependent on the ability to express *disjunctions* of inequalities. In the following sections, we shall see that this is possible in both CP and MIP models.

We can slightly improve the resulting models by eliminating the variables cw_d and ch_d from the model and replacing them by $w_{left(d)}$ and $h_{top(d)}$ in the case that cell d is simple, and so avoiding the need to introduce configuration variables for each simple cell. And if we use the RCSV extension, we can eliminate the variables cw_d and ch_d for all cells d and replace them by $w_{left(d),right(d)}$ and $h_{top(d),bot(d)}$ in the case that cell d spans multiple columns or rows.

4. MIP Encoding

In this section we describe how we have transformed the high-level models in the previous section to models that are suitable for MIP solving techniques. Consider our first and simplest model, the extent table layout model. The only non-linear constraint is $(cw_d, ch_d) \in C_d$.

The *standard encoding* of such a set membership constraint is to introduce a Boolean selector b_k for each configuration, and constrain exactly one selector to be true; this results

in the following constraints for each cell d where $C_d = [(w_1, h_1), ..., (w_N, h_N)]$ is the ordered list of configuration pairs:

$$\begin{aligned} cw_d &= \textstyle\sum_{k=1}^{N} b_k w_k \ \wedge \\ ch_d &= \textstyle\sum_{k=1}^{N} b_k h_k \ \wedge \\ &\textstyle\sum_{k=1}^{N} b_k = 1 \end{aligned}$$

where $b_i \in \{0, 1\}$ for all i.

An alternative approach is use a *unary encoding* of the variable domain. In the unary encoding, we again introduce Boolean selectors to determine the value. However, rather than requiring a single selector to be true, we require an initial sequence to be true; the choice of configuration is determined by the last true selector.

$$\begin{aligned} & cw_d = w_1 + \textstyle\sum_{k=2}^{N} (w_k - w_{k-1}) b_k \\ \wedge \quad & ch_d = h_1 + \textstyle\sum_{k=2}^{N} (h_k - h_{k-1}) b_k \\ \wedge \quad & b_{i-1} \geq b_i, \qquad\qquad i = 3, \ldots, N. \end{aligned}$$

As the configurations are sorted by width, this has the useful property that Boolean b_k is true iff the inequality $cw_d \geq w_k$ is true – b_k can be used to represent the *reified* constraint $cw_d \geq w_k$. This greatly simplifies the construction of complex formulae, such as the disjunctions of inequalities.

In the glue model and the span configurations in the RCSV extension there are additional non-linear constraints of the form $x \in [v_1, \ldots, v_N]$. We can use either the standard or the unary encoding to model these set membership constraints. In the case that we use the unary encoding we again have the property that Boolean b_k effectively represents the reified constraint $x \geq v_k$ since the v_i are in ascending order. For convenience, we will use the notation $[\![x \geq v_i]\!]$ to refer to the Boolean variable representing such an inequality.

Encoding the cell-free model requires us to model disjunctions of the form $ch_d \geq h_i \vee cw_d \geq w_{i+1}$. The straightforward encoding for such a disjunction is to introduce a fresh Boolean variable b and add the constraints

$$ch_d - h_i b \geq 0 \wedge cw_d + w_{i+1} b \geq w_{i+1}$$

Alternatively, if we are using the RCSV extension with the unary encoding we have introduced Boolean variables $[\![ch_d \geq h_i]\!]$ and $[\![cw_d \geq w_{i+1}]\!]$. We can simply re-use these variables and model the disjunction by

$$[\![ch_d \geq h_i]\!] + [\![cw_d \geq w_{i+1}]\!] \geq 1$$

Implementation details

We used a script to construct a mixed integer programming model for each table, which was solved using CPLEX 12.1.

5. Constraint Programming

Constraint programming (e.g. Marriott and Stuckey 1998) is another popular generic approach to solving combinatorial satisfaction problems.

The constraint model is defined in terms of a *domain* of possible values for each variable, and *propagators* for each constraint. The role of a propagator is to remove values from the domains of the variables for that constraint which cannot be part of a solution. Constraint programming can implement combinatorial optimization search by solving a series of satisfaction problems, each time looking for a better solution, until no better solution can be found and optimality is proved.

We consider constraint satisfaction problems, consisting of a set of constraints C over n variables x_i taking integer values, each with a given finite domain $D_{orig}(x_i)$. A feasible solution is a valuation to the variables such that each x_i is within its allowable domain and all constraints are satisfied simultaneously.

A propagation solver maintains a domain restriction $D(x_i) \subseteq D_{orig}(x_i)$ for each variable, and considers only solutions that lie within $D(x_1) \times D(x_2) \times \ldots \times D(x_n)$. Propagators for the constraints C determine, given the current domain, whether we can remove values that cannot take part in any solution. For example, if $x_1 \in \{1,2,3\}$ and $x_2 \in \{2,3\}$ and $C = \{x_1 \geq x_2\}$ then the value $x_1 = 1$ cannot be part of any solution, so it can be eliminated. Propagation solving interleaves propagation, which repeatedly applies propagators to remove unsupported values until no further domain reduction is detected, and search which (typically) splits the domain of some variable in two and considers both the resulting sub-problems. This continues until all variables are fixed and a solution found, or propagation detects *failure* (a variable with empty domain) in which case execution backtracks and tries another subproblem.

Lazy clause generation (Ohrimenko et al. 2009) is a hybrid approach to combinatorial optimization combining finite domain propagation and Boolean satisfiability methods. A lazy clause generation solver performs finite domain propagation just as in a standard CP solver, but records the reasons for all propagations. When a failure is determined it

determines a minimal set of reasons that have caused this failure and records this as a *nogood* in the solver. This nogood prevents the search from examining similar sets of choices which lead to the same inability to solve the problem.

Lazy clause generation is implemented by defining an alternative model for the domains $D(x_i)$, which is maintained simultaneously. Specifically, Boolean variables are introduced for each potential value of a variable, named $[\![x_i = j]\!]$ and similarly $[\![x_i \geq j]\!]$. Negating them gives the opposite, $[\![x_i \neq j]\!]$ and $[\![x_i \leq j-1]\!]$. Fixing such a *literal* modifies D to makes the corresponding fact true in $D(x_i)$ and vice versa. Hence these literals give an alternate Boolean representation of the domain.

In a lazy clause generation solver, the actions of propagators (and search) to change domains are recorded in an *implication graph* over the literals. Whenever a propagator f changes a domain it must *explain* how the change occurred in terms of literals. That is, each literal l that is made true must be explained by a clause $L \rightarrow l$ where L is a conjunction (or set) of literals. When the propagator causes failure it must explain the failure as a *nogood*, $L \rightarrow \mathit{false}$, where L is a conjunction of literals which cannot hold simultaneously. Note that each explanation and nogood is a clause. The explanations of each literal and failure are recorded in the implication graph by introducing an edge to l from each literal used to infer l (that is, an edge from l' to l for each $l' \in L$).

The implication graph is used to build a nogood that records the reason for search failure. We explain the First Unique Implication Point (1UIP) nogood (Moskewicz et al. 2001), which is standard. Starting from the initial failure nogood, a literal l (explained by $L \rightarrow l$) is replaced in the nogood by L by resolution. This continues until there is at most one literal in the nogood made true after the last decision. The resulting nogood is *learnt*, i.e. added as a clause to the constraints of the problem. It will propagate to prevent search trying the same subsearch in the future.

Lazy clause generation effectively imports Boolean satisfiability (SAT) methods for search reduction into a propagation solver. The learnt nogoods can drastically reduce the search space, depending on how often they are reused (i.e. propagated). Lazy clause generation can also make use of SAT search heuristics such as *activity-based search* (Moskewicz et al. 2001). In activity-based search each literal seen in the conflict generation process has its activity bumped, and periodically all activities are decayed. Search decides a literal with maximum activity, which tends to focus on literals that have recently caused failure.

The reason for considering that lazy clause generation might be so effective for the table layout problem is the small number of key decisions that need to be made. While there may be $O(nm)$ cells each of which needs to have an appropriate configuration determined for it, there are only n widths and m heights to decide. These variables define all communication between the cells. Hence if we learn nogoods about combinations of column widths and row heights there are only a few variables involved, and these nogoods are likely to be highly reusable.

It is straightforward to encode the models given in Section 3 using constraint programming. Constraint programming is more expressive than MIP, allowing the use of non-linear constraints. Particularly convenient is that constraint programming solvers typically provide `table` as a built-in global constraint, with the following semantics:

$$\texttt{table}((x_1,\ldots,x_m),i,[(v_{11},\ldots,v_{m1}),\ldots,(v_{1n},\ldots,v_{mn})])$$
$$\equiv$$
$$1 \leq i \leq n \wedge \forall k \in [1,n] \;.\; i = k \Rightarrow x_1 = v_{1k} \wedge \ldots \wedge x_m = v_{mk}$$

That is, it requires x to be the i^{th} element of $[v_1,\ldots,v_n]$. We can then encode the configuration constraints $(cw_d, ch_d) \in \{(w_1,h_1),\ldots,(w_n,h_n)\}$ directly as:

$$\exists i \;.\; \texttt{table}([cw_d, ch_d], i, [(w_1,h_1)\ldots,(w_n,h_n)])$$

Constraint programming solvers also provide reified versions of most primitive constraints which make it straightforward to model disjunction.

Implementation details

For the constraint programming approaches we used the Chuffed lazy clause generation solver. Chuffed is a state-of-the-art CP solver, which scored the most points in all categories of the 2010 MiniZinc Challenge (MiniZinc Challenge 2010) which compares CP solvers.

6. Evaluation

In this section, we evaluate the impact of these modelling techniques when applied to each class of solver. We then evaluate the behaviour of the best model for each solver as the problem size increases. All experiments are performed on a 3.0GHz Core2 Duo with 4Gb RAM running Ubuntu 10.04. For the MIP models, we do not include preprocessing time (to convert the table into a linear program) in the reported runtimes. The models included in the evaluation are outlined in Table 1. All CP models are unary-encoded.

model	description
MIP_{base}	Basic extent model (standard-encoded)
MIP_{r}	Extent model with RCSV
MIP_{ord}	Extent model with unary encoding
MIP_{glue}	Glue model
MIP_{r+p}	Positional encoding with RCSV
MIP_{cf}	Cell-free model (unary-encoded, incl. RCSV)

model	description
CP	Basic extent model (unary-encoded)
CP_{r}	Extent model with RCSV
CP_{r+p}	Positional encoding with RCSV
CP_{r+b}	Position-block encoding with RCSV
CP_{cf}	Cell-free model (incl. RCSV)

Table 1 Description of the evaluated models.

time (s)	MIP_{base}	MIP_{r}	MIP_{ord}	MIP_{glue}	MIP_{r+p}	MIP_{cf}
≤ 0.01	**973**	778	900	674	809	949
≤ 0.10	1158	1120	1167	1052	1130	**1236**
≤ 1.00	1234	1232	1241	1247	1239	**1270**
≤ 10.00	1268	1269	1267	**1271**	**1271**	**1271**
> 10.00	3	2	4	**0**	**0**	**0**

Table 2 Comparison of MIP models on the web-simple data-set. We give the number of instances solved within each time limit.

We first evaluate the model refinements using the corpus of real-world tables described in Gange et al. (2011). This corpus was created by crawling more than 10,000 web pages, then extracting non-nested tables (nested tables are discussed in Section 7), resulting in over 50,000 tables. To choose the goal width for each table, we laid out each web page for three viewport widths (760px, 1000px and 1250px) intended to correspond to common window widths. We then discarded any instances for which the HTML layout algorithm found the optimal solution. This left 2063 table layout problems in the original corpus.

The corpus is partitioned into sets **web-simple** and **web-compound**, based on whether the given table contains any column or row spans. We have expanded the **web-compound** data-set with an additional 231 instances not present in the original evaluation (Gange et al. 2011).

Tables 2 and 3 compare the performance of different refinements to the MIP model. Perhaps surprisingly, although RCSV (MIP_{r}) provides a slight performance improvement in most cases, in cases where the set of configurations are quite dense, the additional constraints can introduce considerable overhead without reducing the search space. Indeed, none of the other individual refinements provide a substantial improvement beyond the base model. However, the combination of the unary encoding and RCSV with the cell-free encoding produces a model (MIP_{cf}) that performs substantially better than any of the component models.

A comparison of the different CP models on the real world data-set is given in Tables 4 and 5. We evaluate the models only using a lazy clause generation solver; a classical non-

time (s)	MIP_{base}	MIP_{r}	MIP_{ord}	MIP_{glue}	MIP_{r+p}	MIP_{cf}
≤ 0.01	756	706	**771**	564	675	766
≤ 0.10	916	874	927	843	881	**964**
≤ 1.00	989	987	997	977	991	**1010**
≤ 10.00	1007	1007	1009	1005	1008	**1013**
> 10.00	6	6	4	8	5	**0**

Table 3 Comparison of MIP models on the web-compound data-set. We give the number of instances solved within each time limit.

time (s)	CP	CP_{r}	CP_{r+p}	CP_{r+b}	CP_{cf}
≤ 0.01	1091	1184	1045	1201	**1258**
≤ 0.10	1219	1259	1224	1264	**1271**
≤ 1.00	1260	**1271**	1268	**1271**	**1271**
≤ 10.00	**1271**	**1271**	**1271**	**1271**	**1271**
> 10.00	**0**	**0**	**0**	**0**	**0**

Table 4 Comparison of CP models on the web-simple data-set. We give the number of instances solved within each time limit.

time (s)	CP	CP_{r}	CP_{r+p}	CP_{r+b}	CP_{cf}
≤ 0.01	665	841	726	867	**952**
≤ 0.10	914	954	946	995	**1004**
≤ 1.00	962	1001	1003	1012	**1013**
≤ 10.00	975	1006	1008	**1013**	**1013**
> 10.00	38	7	5	**0**	**0**

Table 5 Comparison of CP models on the web-compound data-set. We give the number of instances solved within each time limit.

learning CP approach was evaluated in Gange et al. (2011) and found to be totally non-competitive. The basic model successfully solves all of the **web-simple** instances; however, it performs poorly on some of the harder **web-compound** instances. Augmenting this model with RCSV (CP_{r}) provides a performance improvement; however, it still fails to solve some instances which have many symmetric solutions. Using a positional encoding (CP_{r+p}) solves several of these instances, but causes poor performance on some instances with large numbers of rows (and incurs an overhead on the non-compound instances). Using the position-block encoding (CP_{r+b}) allows us to combine the advantages of both encodings; it solves all **web-compound** instances, and doesn't incur an overhead on the long simple tables. Combining this with the cell-free model (CP_{cf}) is clearly the best encoding overall.

Tables 6 and 7 collate the results for CP and MIP_{glue} – the best models from Bilauca and Healy (2010, 2011), Gange et al. (2011) – and the improved cell-free models. Table 6 gives the results of the selected methods on the **web-simple** dataset. All the selected methods solve all instances within the 10 second time-limit; and the cell-free CP model clearly outperforms all the other methods, solving all instances in no more than 0.1 seconds.

time (s)	CP	CP_{cf}	MIP_{glue}	MIP_{cf}
≤ 0.01	1091	**1258**	674	949
≤ 0.10	1219	**1271**	1052	1236
≤ 1.00	1260	**1271**	1247	1270
≤ 10.00	**1271**	**1271**	**1271**	**1271**
> 10.00	**0**	**0**	**0**	**0**

Table 6 Number of instances from the web-simple data-set solved within each time limit by selected methods.

time (s)	CP	CP_{cf}	MIP_{glue}	MIP_{cf}
≤ 0.01	665	**952**	564	766
≤ 0.10	914	**1004**	843	964
≤ 1.00	962	**1013**	977	1010
≤ 10.00	975	**1013**	1005	**1013**
> 10.00	38	**0**	8	**0**

Table 7 Number of instances from the web-compound data-set solved within each time limit by selected methods.

Results on **web-compound** are given in Table 7. Only the two cell-free models solved all instances in less than 10 seconds, with CP_{cf} again being the fastest method. The poor performance of the original CP model illustrates the impact of the extent encoding in the presence of compound cells; in many of the failed instances, the CP model quickly finds the optimum solution, but must test a large number of symmetric solutions to prove optimality. MIP does not suffer from the same difficulties, as the linear relaxation quickly determines that the symmetric solutions cannot improve the objective function. Indeed, there is very little performance difference between MIP_{cf} using the extent encoding and the position-block encoding.

While all the methods perform well on the real-world instances, we are also interested in how the solver performance scales. We generated a set of artificial tables to test the solver performance as the number of rows, number of columns, and frequency of compound cells increases. Artificial $n \times m$ tables were generated by selecting a random k-word piece of text for each cell, where k is chosen from a normal distribution with $\mu = 6$, $\sigma = 3$ (with a minimum of 1 word per cell) – the text is taken from the Project Gutenberg edition of The Trial (Kafka 1925, 2005). Let $minW$ (resp. $maxW$) be the width of the table when each cell is assigned its narrowest (widest) configuration. We then define the *squeeze* of a width W as $\frac{W-minW}{maxW}$. For these experiments, we selected a squeeze of 0.25. Given the considerable difference in performance between methods, all times are shown on a log-scale.

Figure 5 gives the results on $r \times 10$ tables as r is varied between 10 and 200. The MIP approaches clearly scale better on these instances than the other methods. This is likely

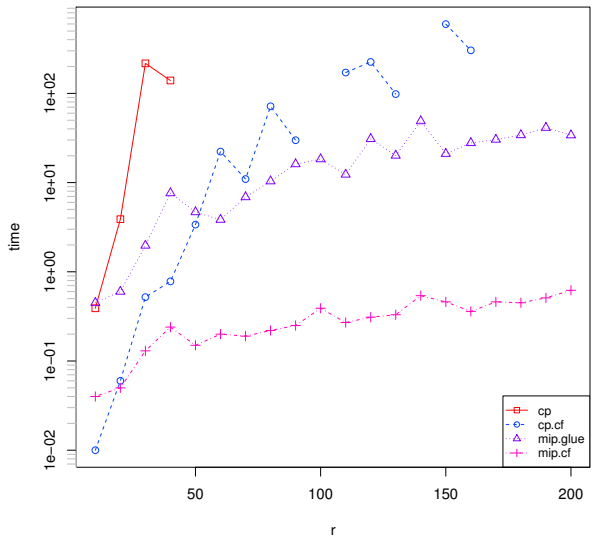

Figure 5 **Results for artificially constructed $r \times 10$ tables. Times are in seconds. Missing entries indicate a time-out.**

because, although the number of rows increases, the number of variables that are binding on the width constraint remains the same.

Figure 6 shows performance on $10 \times c$ tables. In this case, the two cell-free models scale considerably better than the other methods. Interestingly, CP_{cf} solves these instances slightly faster than MIP_{cf}. This is likely because, with only 10 rows, both methods can prove optimality relatively easily; and the lazy clause generation solver has slightly less overhead when propagating values across the cell configuration constraints.

We also tested performance on tables of a fixed size as the number of compound cells was increased. As the cell-free models solve 10×10 tables too quickly to give meaningful results, we constructed 20×20 tables. Text for a $w \times h$ compound cell is again selected from a normal distribution, but with $\mu = 6wh$. In this case, there isn't a uniform increase in difficulty; although introducing compound cells introduces a more complex structure to the table, it also reduces the overall number of cells. Nevertheless, the cell-free models are uniformly the most robust approaches, both MIP_{cf} and CP_{cf} being competitive.

7. Extensions

Up until now we have only considered layout of a flat table without any designer constraints. In this section we investigate how our approaches can be extended to handle nested tables and designer constraints.

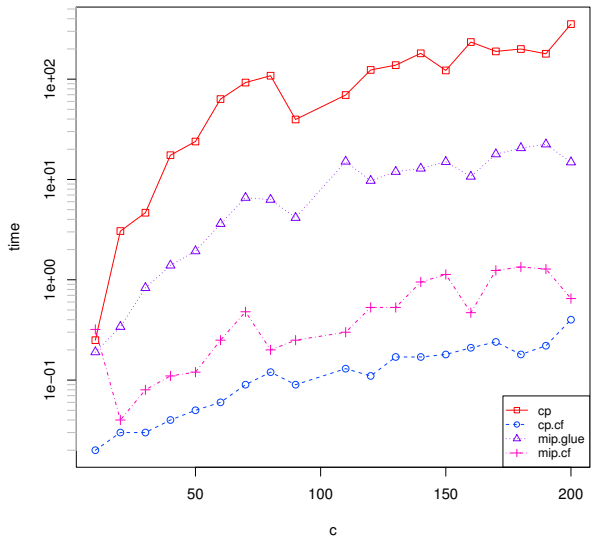

Figure 6 **Results for artificially constructed $10 \times c$ tables. Times are in seconds. Missing entries indicate a time-out.**

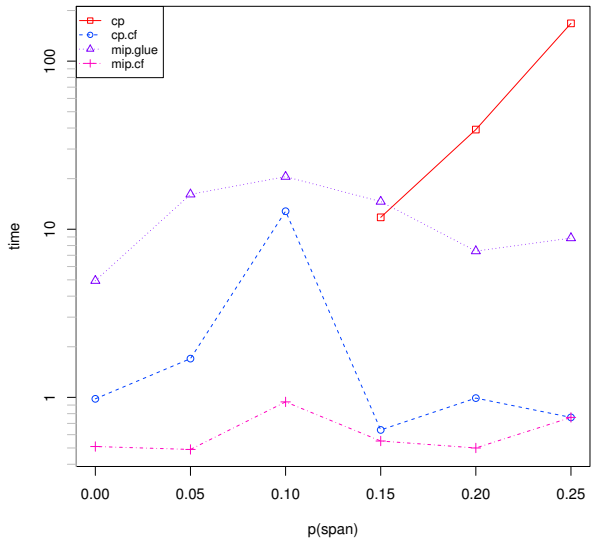

Figure 7 **Results for artificially constructed 20×20 tables with an increasing proportion of compound cells. Missing entries indicate a time-out.**

As we saw in Section 3 designer constraints are specific to a table and useful designer constraints specify that column widths or row heights are fixed ratios of each other or the overall table width, or have a fixed size. Such designer constraints give rise to simple linear

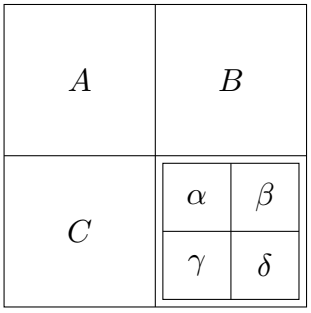

Figure 8 **A small instance of a nested table. A, B and C are primitive cells in the outer table. Cell D contains a sub-table with cells α through δ.**

constraints over the column widths or row heights so are readily handled using generic solving techniques such as MIP or CP.

In practice many tables contain other tables. HTML tables, for example, allow table cells to contain arbitrary HTML elements which can be tables or text or a mixture of both. The simplest approach is to perform table layout recursively, using the table layout algorithm to produce a set of minimal width/height configurations for the innermost tables, and then work outwards at each stage having the complete set of minimal configurations for the table's cells. We call this the *recursive* approach. The main disadvantage is that the number of minimal configurations for a table can be very large, meaning that this approach is expensive.

Usually in nested tables the cells do not contain arbitrary mixtures of tables and other content. Much more commonly they follow the rule that a table cell either contains a nested table and nothing else or it contains HTML content but no table. In this case we can solve the nested table by *flattening* the nested tables to give a single non-recursive optimisation problem. If cell d contains a table T^d with widths w^d and heights h^d then we recursively generate the constraints for T^d except for the maximum width requirement and also add the constraints

$$cw_d = wd_{1,n_T}(w) \ \wedge \ ch_d = ht_{1,m_T}(h)$$

to constrain the configuration for cell d to be a layout for T^d. The height of the outermost table is then minimised with respect to the complete set of generated constraints.

EXAMPLE 6. Consider the small nested table shown in Figure 8. The constraints generated for the inner table are:

$$\begin{aligned}
& (cw_d, ch_d) \in C_d, && \forall d \in \{\alpha, \beta, \gamma, \delta\} \\
\wedge\ & ht_{1,1}(h^D) \geq ch_d, && \forall d \in \{\alpha, \gamma\} \\
\wedge\ & ht_{2,2}(h^D) \geq ch_d, && \forall d \in \{\beta, \delta\} \\
\phi_D = \wedge\ & wd_{1,1}(w^D) \geq cw_d, && \forall d \in \{\alpha, \beta\} \\
\wedge\ & wd_{2,2}(w^D) \geq cw_d, && \forall d \in \{\gamma, \delta\} \\
\wedge\ & ch_D = ht_{1,2}(h^D) \\
\wedge\ & cw_D = wd_{1,2}(w^D)
\end{aligned}$$

Once we have variables (cw_D, ch_D) for the dimensions of the inner table, we can construct the model for the outer table as usual:

$$\begin{aligned}
& \text{find } w \text{ and } h \text{ that minimize } ht_{1,2}(h) \text{ subject to} \\
& (cw_d, ch_d) \in C_d, && \forall d \in \{A, B, C\} \\
\wedge\ & ht_{1,1}(h) \geq ch_d, && \forall d \in \{A, C\} \\
\wedge\ & ht_{2,2}(h) \geq ch_d, && \forall d \in \{B, D\} \\
\wedge\ & wd_{1,1}(w) \geq cw_d, && \forall d \in \{A, B\} \\
\wedge\ & wd_{2,2}(w) \geq cw_d, && \forall d \in \{C, D\} \\
\wedge\ & \phi_D \\
\wedge\ & wd_{1,2}(w) \leq W
\end{aligned}$$

□

It is straightforward to solve the flattened set of constraints using either MIP or CP techniques. All the modelling techniques described in Section 3 can also be applied to problems with nested tables, with the exception that RCSV cannot be applied to spans containing sub-tables.

8. Conclusion

Treating table layout as a constrained optimization problem allows us to use powerful generic approaches to combinatorial optimization to solve these problems to optimality. We have given a variety of models for table layout and evaluated these using both MIP and constraint programming with lazy clause generation implementations.

Our first empirical evaluation used a corpus of over 2,000 HTML tables collected from the Web that were hard to solve in the sense that the standard HTML table layout algorithm

did not find the minimal height layout. We found that all methods worked quite well and solved almost all problems in less than 1 second, the cell-free CP model being uniformly fastest.

In our second empirical evaluation we "stress-tested" the best methods from the previous evaluation using artificial table layout examples of increasing number of columns, rows or percentage of compound cells. In this case we again found the cell-free encodings dominated, the MIP model being slightly more robust on these artificial instances.

Both approaches can be easily extended to handle designer constraints on table widths such as enforcing a fixed size or that two columns must have the same width. They can also be extended to a simple form of nested tables, where the cell contents are allowed to either be a table or text, but not a combination. The case when complex combinations of text and tables are allowed is more difficult, and is something we plan to pursue.

9. Acknowledgements

The Australian-based authors acknowledge the support of the ARC through Discovery Project Grant DP0987168

References

Anderson, Richard J., Sumeet Sobti. 1999. The table layout problem. *SCG '99: Proceedings of the Fifteenth Annual Symposium on Computational Geometry*. ACM Press, New York, NY, USA, 115–123. doi: http://doi.acm.org/10.1145/304893.304937.

Badros, Greg J., Alan Borning, Kim Marriott, Peter Stuckey. 1999. Constraint cascading style sheets for the web. *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*. ACM, New York, 73–82.

Beach, Richard John. 1985. Setting tables and illustrations with style. Ph.D. thesis, University of Waterloo.

Beaumont, Nicholas. 2004. Fitting a table to a page using non-linear optimization. *Asia-Pacific Journal of Operational Research* **21** 259–270.

Bilauca, Mihai, Patrick Healy. 2010. A new model for automated table layout. *Proceedings of the 10th ACM symposium on Document engineering*. DocEng '10, ACM, 169–176.

Bilauca, Mihai, Patrick Healy. 2011. Building table formatting tools. *Proceedings of the 11th ACM symposium on Document engineering*. ACM, 13–22.

Borning, Alan, Richard Lin, Kim Marriott. 2000. Constraint-based document layout for the web. *Multimedia Systems* **8** 177–189.

Gange, G., K. Marriott, P. Moulder, P. Stuckey. 2011. Optimal automatic table layout. *Proceedings of the 11th ACM Symposium on Document Engineering*. 23–32.

Hurst, N., W. Li, K. Marriott. 2009. Review of automatic document formatting. *Proceedings of the 9th ACM symposium on Document engineering*. ACM, 99–108.

Hurst, Nathan, Kim Marriott, David Albrecht. 2006. Solving the simple continuous table layout problem. *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*. ACM, New York, NY, USA, 28–30. doi:http://doi.acm.org/10.1145/1166160.1166169.

Hurst, Nathan, Kim Marriott, Peter Moulder. 2005. Towards tighter tables. *Proceedings of Document Engineering, 2005*. ACM, New York, 74–83.

Kafka, Franz. 1925, 2005. *The Trial*. Project Gutenberg.

Lin, X. 2006. Active layout engine: Algorithms and applications in variable data printing. *Computer-Aided Design* **38** 444–456.

Lutteroth, Christof, Gerald Weber. 2006. User interface layout with ordinal and linear constraints. *AUIC '06: Proceedings of the 7th Australasian User interface conference*. Australian Computer Society, Inc., 53–60.

Marriott, Kim, Peter Moulder, Nathan Hurst. 2013. Html automatic table layout. *ACM Transactions on the Web (TWEB)* **7** 4.

Marriott, Kim, Peter Stuckey. 1998. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, Massachusetts.

MiniZinc Challenge. 2010. 2010 MiniZinc challenge. `http://www.g12.csse.unimelb.edu.au/minizinc/challenge2010/results2010.html`.

Moskewicz, Matthew W., Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik. 2001. Chaff: Engineering an efficient SAT solver. *DAC '01: Proceedings of the 38th conference on Design automation*. 530–535. doi:10.1145/378239.379017.

Ohrimenko, O., P.J. Stuckey, M. Codish. 2009. Propagation via lazy clause generation. *Constraints* **14** 357–391.

Raggett, Dave, Arnaud Le Hors, Ian Jacobs. 1999. HTML 4.01 Specification, section 'Autolayout Algorithm'. http://www.w3.org/TR/html4/appendix/notes.html#h-B.5.2.

Wang, Xinxin, Derick Wood. 1997. Tabular formatting problems. *PODP '96: Proceedings of the Third International Workshop on Principles of Document Processing*. Springer-Verlag, London, UK, 171–181.