Optimal Automatic Table Layout

Graeme Gange
Dept of CSSE
University of Melbourne
Vic. 3010, Australia
ggange@csse.unimelb.edu.au

Kim Marriott and Peter Moulder
Clayton School of IT
Monash University
Vic. 3800, Australia
{kim.marriott, peter.moulder}@monash.edu

Peter Stuckey
Dept of CSSE
University of Melbourne
Vic. 3010, Australia
pjs@csse.unimelb.edu.au

ABSTRACT

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 the viewport width, choice of font and dynamic content. However, if the table contains text, minimizing the height of the table for a fixed maximum width is a difficult combinatorial optimization problem. We present three different approaches to finding the minimum height layout based on standard approaches for combinatorial optimization. All are guaranteed to find the optimal solution. The first is an A*-based approach that uses an admissible heuristic based on the area of the cell content. The second and third are constraint programming (CP) approaches using the same CP model. The second approach uses traditional CP search, while the third approach uses a hybrid CP/SAT approach, lazy clause generation, that uses learning to reduce the search required. We provide a detailed empirical evaluation of the three approaches and also compare them with two mixed integer programming (MIP) encodings due to Bilauca and Healy.

Categories and Subject Descriptors

I.7.2 [**Document and Text Processing**]: Document Preparation—*Format and notation, Photocomposition/typesetting*

General Terms

Algorithms

Keywords

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, CSS and XSL. For on-line presentation it is not practical to require the author to specify table column widths at document authoring time since the layout needs to adjust to different width viewing environments and to different sized text since, for instance, the viewer may choose a larger font. Dynamic content is another reason that it can be impossible for the author to fully specify table column widths. This is an issue for web pages and also for VDP in which improvements in printer technology now allow companies to cheaply print material which 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 many 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 expensive. Anderson and Sobti [1] have shown that table layout with text is NP-hard. The reason is that if a cell contains text then this implicitly constrains the cell to take one of a discrete number of possible configurations corresponding to different numbers of lines of text. It is a difficult combinatorial optimization problem to find which combination of these discrete configurations best satisfies reasonable layout requirements such as minimizing table height for a given width.

Table layout research is reviewed in Hurst, Li & Marriott [7]. Starting with Beach [3], a number of authors have investigated automatic table layout from a constrained optimization viewpoint and a variety of approaches for table layout have been developed. Almost all approaches use heuristics and are not guaranteed to find the optimal solution. They include methods that use a desired width for each column and scale this to the actual table width [17, 6, 2], methods that use a continuous linear or non-linear approximation to the constraint that a cell is large enough to contain its contents [1, 4, 9, 8, 11], a greedy approach [9] and an approach based on finding a minimum cut in a flow graph [1].

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. Even if the complete techniques are too slow 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. For example, while Gecko (the layout engine used by the Firefox web browser) is the most sophisticated of the HTML/CSS user agents whose source code we've seen, Figure 1 shows that its layouts can be far

DocEng'11, September 19–22, 2011, Mountain View, California, USA.

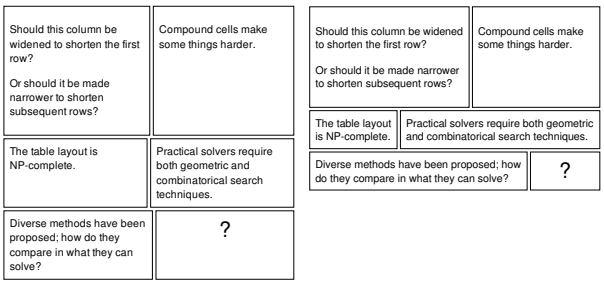

Figure 1: Example table comparing layout using Gecko (on the left) with the minimal height layout (on the right).

from the most compact as computed by one of the algorithms we present.

We know of only two other papers that have looked at complete methods for table layout with breakable text. The first is a branch-and-bound algorithm described in [19], which finds a layout satisfying linear designer constraints on the column widths and row heights. However it is only complete in the sense that it will find a feasible layout if one exists and is not guaranteed to find an optimal layout that, say, minimizes table height.[1] The second is detailed in a recent paper by Bilauca and Healy [5]. They give two MIP based branch-and-bound based complete search for simple tables.

The first contribution of this paper is to present three new techniques for finding a minimal height table layout for a fixed width. All three are 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 an $A^\star$ based approach [18] that chooses a width for each column in turn. Efficiency of the $A^\star$ algorithm crucially depends on having a good conservative heuristic for estimating the minimum height for any full table layout that extends the current layout. We use a heuristic that treats the remaining unfixed columns in the layout as if they are a single merged column each of whose cells must be large enough to contain the contents of the unfixed cells on that row. The other key to efficiency is to prune layouts that are not *column-minimal* in a sense that it is possible to reduce one of the fixed column widths without violating a cell containment constraint while keeping the same row heights.

The second and third approach are constraint programming (CP) [13] approaches. We model the problem using constraint programming and apply two different solving technologies to the same model. The second approach uses traditional CP search, while the third approach uses a hybrid solving approach, lazy clause generation [16], which combines CP and SAT technology. The advantages 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. Both these advantages can drastically reduce the amount of search required.

The second contribution of this paper is to provide an extensive empirical evaluation of these three approaches as well as the two MIP-based approaches of Bilauca and Healy [5]. 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 stress-tested the algorithms on some large artificial table layout examples. In this case we found that the hybrid CP/SAT approach was the most robust approach.

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

2. BACKGROUND

2.1 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 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:

$$\begin{array}{lcllcl} wd_{c_1,c_2}(w) & = & \sum_{c=c_1}^{c_2} w_c, & wd(w) & = & wd_{1,n}(w), \\ ht_{r_1,r_2}(h) & = & \sum_{r=r_1}^{r_2} h_r, & ht(h) & = & ht_{1,m}(h) \end{array}$$

where ht and wd 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 cell* consists of multiple grid elements forming a rectangle, i.e. the grid elements span contiguous rows and columns.

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

$$\begin{array}{ll} bot(d) = \max rows(d), & top(d) = \min rows(d), \\ left(d) = \min cols(d), & right(d) = \max cols(d). \end{array}$$

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

$$\begin{array}{c} rcells_c = \{d \in Cells \mid right(d) = c\}, \\ cells_c = \{d \in Cells \mid c \in cols(d)\}, \\ bcells_r = \{d \in Cells \mid bottom(d) = r\} \end{array}$$

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 text in the cell.

The table's *structural constraints* are that each cell is big enough to contain its content and at least as wide as its minimum width and as high as its minimum height.

The table *layout style* captures what is required in a good layout. We shall focus on the *minimum height* layout style. This finds a layout for the table that, in decreasing order of importance, is no wider than the fixed maximum width, minimizes table height, and minimizes table width.

The problem we are addressing is, given a table structure and content for the table cells, to find an assignment to the column widths and row heights such that the structural constraints are satisfied and the minimum height layout style is satisfied.

For simplicity, we assume that the minimum table width is wide enough to allow the structural constraints to be satisfied. Furthermore, we do not consider nested tables nor do we consider designer constraints such as columns having fixed ratio constraints between them.

2.2 Minimum configurations

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) s.t. 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 minimum configurations are *anti-monotonic* in the sense that if the width increases then the height will never increase. 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 minimum configurations of the text in a cell [7]. Here we assume that these are pre-computed and that

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

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.

The mathematical model of the table layout problem can be formalized as:

find w and h that minimize $ht(h)$ subject to

$$\forall d \in Cells. \quad (cw_d, ch_d) \in configs_d \wedge \quad (1)$$
$$\forall d \in Cells. \quad wd_{left(d),right(d)}(w) \geq cw_d \wedge \quad (2)$$
$$\forall d \in Cells. \quad ht_{top(d),bot(d)}(h) \geq ch_d \wedge \quad (3)$$
$$wd(w) \leq W \quad (4)$$

In essence, automatic table layout is the problem of finding *minimal configurations for a table*: i.e. minimal width / height combinations in which the table can be laid out. One obvious necessary condition for a table layout (w, h) to be a minimal configuration is that it is impossible to reduce the width of any column c while leaving the other row and column dimensions unchanged and still satisfy the structural constraints. We call a layout satisfying this condition *column-minimal*.

We now detail three algorithms for solving the table layout problem. All are guaranteed to find an optimal solution but in the worst case may take exponential time.

3. A* ALGORITHM

The first approach uses an $A^\star$ based approach [18] that chooses a width for each column in turn. A partial layout (w, c) for a table is a width w for the first $c-1$ columns. The algorithm starts from the empty partial layout ($c = 1$) and repeatedly chooses a partial layout to extend by choosing possible widths for the next column.

Partial layouts also have a penalty p, which is a lower bound on the height for any full table layout that extends the current partial layout. The partial layouts are kept in a priority queue and at each stage a partial layout with the smallest penalty p is chosen for expansion. The algorithm has found a minimum height layout when the chosen minimal-penalty partial layout has $c = n + 1$ have at least as great a penalty then all other partial layouts must lead to at least as tall a layout. The code is given in function *complete-A*-search*(W) where W is the maximum allowed table width. For simplicity we assume W is greater than the minimum table width. (The minimum table width can be determined by assigning each column its min_c width from *possible-col-widths*, or can equivalently be derived from the corresponding maximum positions also used in that function.)

Given widths w for columns $1, \ldots, c-1$ and maximum table width of W, the function *possible-col-widths*(c,w,W) returns the possible widths for column c that correspond to the width of a minimal configuration for a cell in c and which satisfy the minimum width requirements for all the cells in d and still satisfy the minimum width requirements for columns $c + 1, \ldots, n$ and allow the table to have width W.

Efficiency of an $A^\star$ algorithm usually depends strongly on how tight the lower bound on penalty is, i.e. how often (and how early) the heuristic informs us that we can discard a partial solution because all full table layouts that extend that partial layout will either have a height greater than the optimal height, or have height greater or equal to some other layout that isn't discarded.

We use a heuristic that treats the remaining unfixed columns in the layout as if they are a single merged column each of whose cells must be large enough to contain the contents of the unfixed cells on that row. We approximate the contents by a lower bound of their area. The function *compute-approx-row-heights*(w,h,c,W) does this, returning the estimated (lower bound) row heights after laying out the area of the contents of columns $c + 1, \ldots, n$ in a single column whose width brings the table width to W. Compound cells that span multiple rows, and positions in the table grid that have no cell, use a very simple lower bound of zero. (A simple refinement would be to use the product of the width requirement for the cell's column span and the height requirement for each row.)

A standard method of discarding partial solutions that must lead to solutions no better than some other partial solution is to use a *closed set*. However, for a standard closed set, the key would include the height requirements of each row span (including non-compound row spans) encountered, as well as certain column start positions. Implementers might consider how useful such a closed set would be for the inputs of interest to them. Our implementation doesn't use one.

We instead present the following more interesting method for discarding partial solutions. Partial layouts which must lead to a full layout which is not column minimal are not considered. If the table has no compound cells spanning multiple rows then any partial layout that is not column minimal for the columns that have been fixed can be discarded because row heights can only increase in the future and so the layout can never lead to a column-minimal layout. This no longer holds if the table contains cells spanning multiple rows as row heights can decrease and so a partial

layout that is not column minimal can be extended to one that is column minimal. However, it is true that if the cells spanning multiple rows are ignored, i.e. assumed to have zero content, when determining if the partial layout is column minimal then partial layouts that are not column minimal can be safely discarded. The function *weakly-column-minimal*(w,c) does this by checking that none of the columns $1, \ldots, c$ can be narrowed without increasing the height of a row, ignoring compound cells spanning multiple rows.

In our implementation of *complete-A*-search*, the iteration over possible widths works from maximum v downwards, stopping once the new partial solution is either known not to be column minimal or (optionally) once the penalty exceeds a certain maximum penalty which should be an upper bound on the minimum height. Our implementation computes a maximum penalty at the start, by using a heuristic search based on [9].

Creating a new partial layout is relatively expensive (see below), so this early termination is more valuable than one might otherwise expect. However, the cost of this choice is that this test must be done before considering the height lower bounds for future cells (the remaining-area penalty), since the future penalty is at its highest for maximum v.

For the implementation of *compute-approx-row-heights*, note that $D2_r$, $area_r$ and the c' bound in the sum don't depend on w or $h0$, and hence may be calculated once off in advance; while w may be stored in cumulative form; so that the loop body can run in constant time. (Our implementation uses this approach.)

Our implementation literally evaluates *minheight* once per cell ending at c for each partial solution being added to the queue, as Figure 2 depicts. One could instead combine some of this work with finding the union of widths of interest (the second half of *possible-col-widths*), by changing the sorted list of configurations of interest for a cell to store the increase in height for its row span, so that *complete-A*-search* can update the height requirements for the current column (and in turn update the height requirements for the new partial solution) as it iterates over the combined list of widths. How significant a saving this would be would depend on how expensive the *minheight* implementation is compared to preparing and storing the new partial solution's height requirements. For it to have a significant payoff might require changing the representation of height requirements to avoid copying the full set for each new partial solution.

For *possible-col-widths*, our current implementation of the min_c calculation literally iterates over all $rcells_c$, though one could instead calculate in advance a single *minw*-like value for each column span occurring in the table, and have the min_c calculation iterate over the set of column spans ending at c (which would often be a singleton set) instead of all the cells ending at c. Less valuably, one could similarly calculate in advance a set of possible widths for each column span, and have the $widths_d$ calculation iterate over the set of column spans ending at c instead of all cells ending at c. (Again, our current implementation does not do this.)

Whereas for the max_c calculation, we do calculate in advance a single array of maximum positions (one per column), so max_c is a simple lookup.

For *weakly-column-minimal*, our implementation is comparable to what's written in the figure: we do cache h, but we still iterate over all the cells placed so far to determine column minimality. In principle, one can do better than this, by keeping track of what height a row span must increase to allow narrowing a cell having that row span and being one of the cells forcing a column end position to be considered minimal. We haven't looked into this, though looking more might be worthwhile for uses where difficult, wide tables are common.

4. CONSTRAINT PROGRAMMING

Constraint programming (see e.g. [13]) is an approach to combinatorial satisfaction problems which combines search and inference. The constraint model is defined in terms of a *domain* of possible values for each variables, 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 the optimal 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, e.g. 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 [16] 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

function *possible-col-widths*(c,w,W)
$min_c := \max_{d \in rcells_c} \{minw(d) - wd_{left(d),c-1}(w)\}$
for $c' := n$ down to $c+1$ **do**
  $w_{c'} := \max_{d \in lcells_{c'}} \{minw(d) - wd_{c'+1,right(d)}(w)\}$
endfor
$max_c := W - wd_{1,c-1}(w) - wd_{c+1,n}(w)$
for $d \in rcells_d$ **do**
  $widths_d := \{w_k - wd_{left(d),c-1}(w) | (w_k, h_k) \in configs_d\}$
  $widths_d := \{v \in widths_d \mid min_c \le v \le max_c\}$
return $(\bigcup_{d \in rcells_d} widths_d)$

function *weakly-column-minimal*(w,c)
for $r := 1$ to m **do**
  $D_r := \{d \in Cells \mid right(d) \le c \text{ and } rows(d) = \{r\}\}$
  $h_r := \max_{d \in D_r} \{minheight(d, wd_{left(d),right(d)}(w))\}$
endfor
for $c' := 1$ to c **do**
  $cm := false$
  for $d \in rcells_{c'}$ s.t. $|rows(d)| = 1$ **do**
    if $minheight(d, wd_{left(d),c'}(w) - \epsilon) > h_{bot(d)}$
    then $cm := true$; break
  endfor
  if not cm **then return false**
endfor
return true

function *compute-approx-row-heights*(w,$h0$,c,W)
for $r := 1$ to m **do**
  $D1_r := \{d \in Cells \mid right(d) = c \text{ and } bot(d) = r\}$
  if $D1_r = \emptyset$ **then** $h1 := 0$
  else $h1 := \max_{d \in D1_r} \{ minheight(d, wd_{left(d),right(d)}(w)) - ht_{top(d),r-1}(h) \}$
  endif
  $D2_r := \{d \in Cells \mid c < right(d) \text{ and } rows(d) = \{r\}\}$
  $area_r := \sum_{d \in D2_r} area(d)$
  if $area_r = 0$ **then** $h2 := 0$
  else $h2 := area_r / \left(W - \sum_{c'=1}^{\min(\{c\} \cup \{left(d) | d \in D2_r\})} w_{c'} \right)$
  endif
  $h_r := \max\{h0_r, h1, h2\}$
endfor
return h

function *complete-A*-search*(W)
create a new priority-queue q
add $(0, -1, [c \mapsto 0 | c = 1..n], [r \mapsto 0 | r = 1..m])$ to q
repeat
  remove lowest priority state $(p, -c, w, h)$ from q
  if $c = n+1$ **then return** (w, h)
  $widths_c := $ *possible-col-widths*(c, w, W)
  foreach $v \in widths_c$ s.t. *weakly-column-minimal*$(w[c \mapsto v], c)$ **do**
    $w' := w[c \mapsto v]$
    $h' :=$ *compute-approx-row-heights*(w,h,c,W)
    add $(ht(h'), -(c+1), w', h')$ to q
  endfor
forever

Figure 2: An A* algorithm for table layout.

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 (set or) conjunction of literals. When the propagator causes failure it must explain the failure as a *nogood*, $L \rightarrow false$, 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 with edges from each $l' \in L$ to 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 [15], 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. propagate). Lazy clause generation can also make use of SAT search heuristics such as *activity-based search* [15]. 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.

4.1 A CP model for table layout

A Zinc [12] model is given below. Each cell d has a configuration variable $f[d]$ which chooses the configuration (cw, ch) from an array of tuples $cf[d]$ of (width, height) configurations defining $configs_d$. Note that $t.1$ and $t.2$ return the first and second element of a tuple respectively. The important variables are: w, the width of each column, and h, the height of each row. These are constrained to fit each cell, and so that the maximum width is not violated.

```
int: n; % number of columns
int: m; % number of rows
int: W; % maximal width
set of int: Cells; % numbered cells
array[Cells] of 1..m: top;
array[Cells] of 1..m: bot;
array[Cells] of 1..n: left;
array[Cells] of 1..n: right;
array[Cells] of array[int] of tuple(int,int): cf;

array[Cells] of var int: f; % cell configurations
array[1..n] of var int: w;  % column widths
array[1..m] of var int: h;  % row heights

constraint forall(d in Cells)(
      % constraint (2)
  sum(c in left[d]..right[d])(w[c])>=cf[d][f[d]].1
  /\  % constraint (3)
  sum(r in top[d]..bot[d])(h[r])>=cf[d][f[d]].2
);
      % constraint (4)
constraint sum(c in 1..n)(w[c]) <= W;
solve minimize sum(r in 1..m)(h[r]);
```

The Zinc model does not enforce column minimality of the solutions, but solutions will be column minimal because of the optimality condition.

The reason that we thought 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. We can see the benefit of nogood learning by comparing the constraint programming model, with and without learning.

EXAMPLE 4.1. *Consider laying out a table of the form*

```
aa      aa
    aa      aa  aa
        aa  aa
            aa  aa
                aa  aa
            aa      aa
```

where each aa entry can have two configurations: wide *two characters wide and one line high, or* high *one character wide and two lines high (so* `cf[d] = [(2,1),(1,2)]`*). Assume the remaining cells have unique configuration (1,1), and there is a maximal table width of 9, and a maximal table height of 9. Choosing the configuration of cell (1,1) as* wide (`f[(1,1)] = 1`*) makes $w_1 \geq 2$, similarly if cell (1,3) is* wide *then $w_3 \geq 2$. The effect of each decision in terms of propagation is illustrated in the implication graph in Figure 3 Now choosing the configuration of cell (2,2) as* wide *makes $w_2 \geq 2$ and then propagation on the sum of column widths forces each of the remaining columns to be at most width 1: $w_4 \leq 1, w_5 \leq 1, w_6 \leq 1$. Then $w_4 \leq 1$ means $h_2 \geq 2$, $h_3 \geq 2$, $h_4 \geq 2$ and $h_6 \geq 2$ since we must pick the second configuration for each of the cells in column 4. These height constraints together violate the maximal height constraint. Finite domain propagation backtracks undoing the last decision and sets the configuration of (2,2) as* high (`f[(2,2)] = 2`*), forcing $h_2 \geq 2$. Choosing the configuration of (2,5) as* wide *makes $w_5 \geq 2$ and then propagation on the sum of column widths forces each of the remaining columns to be at most width 1: $w_2 \leq 1, w_4 \leq 1, w_5 \leq 1$. Again $w_4 \leq 1$ means the maximal height constraint is violated. So search undoes the last decision and sets the configuration of (2,5) as* high.

Lets contrast this with lazy clause generation. After making the first three decisions the implication graph is shown in Figure 3. The double boxed decisions reflect making the cells (1,1), (1,3) and (2,2) wide. The consequences of the last decision are shown in dashed boxes. Lazy clause generation starts from the nogood $h_2 \geq 2 \wedge h_3 \geq 2 \wedge h_4 \geq 2 \wedge h_6 \geq 2 \rightarrow false$ and replaces $h_6 \geq 2$ using its explanation $f[(6,4)] = 2 \rightarrow h_6 \geq 2$ to obtain $h_2 \geq 2 \wedge h_3 \geq 2 \wedge h_4 \geq 2 \wedge f[(6,4)] = 2 \rightarrow false$. This process continues until it arrives at the nogood $w_4 \leq 1 \rightarrow false$ which only has one literal from the last decision level. This is the 1UIP nogood. It will immediately backjump to the start of the search (since the nogood does not depend on any other literals at higher decision levels) and enforce that $w_4 \geq 2$. Search will again make cell (1,1) wide*, and on making cell (1,3)* wide *it will determine $w_3 \geq 2$ and consequently that $w_2 \leq 1$, $w_5 \leq 1$ and $w_6 \leq 1$ which again causes violation of the maximal height constraint. The 1UIP nogood is $w_1 \geq 2 \wedge w_3 \geq 2 \rightarrow fail$, so backjumping removes the last choice and infers that $w_3 \leq 1$ which makes $h_1 \geq 2$ and $h_3 \geq 2$.*

Note that the lazy clause generation completely avoids considering the set of choices (1,1), (1,3) and (2,5) wide *since it already fails on setting (1,1) and (1,3)* wide*. This illustrates how lazy clause generation can reduce search. Also notice that in the implication graph the consequences of a configuration choice only propagate through width and height variables, and hence configuration choices never appear in nogoods.*

5. MIXED INTEGER PROGRAMMING

Bilauca and Healy [5] consider using mixed integer programming (MIP) to model the table layout problem. They consider two models for the simple table layout problem and do not consider compound cells, i.e. row and column spans. Their basic model BMIP uses 01 variables (`cellSel`) for each possible configuration, to model the integer configuration choice `f` used in the CP model. This basic model can be straightforwardly extended to handle compound cells.

Their improved model adds redundant constraints on the column widths to substantially improve MIP solving times for harder examples. They compute the minimum width (`minW`) for each column as the maximum of the minimum widths of the cells in the column, and the minimum height (`minH`) for each row analogously. They then compute the set of possible column widths (`colWSet`) for each column from those configurations in the column which have at least width `minW` and height `minH`. Note this improvement relies on the fact that there are no column spans. While in practice this usually does provide the set of possible widths in any column-minimal layout, in general we believe the "improved model" is incorrect.

EXAMPLE 5.1. *Consider laying out a 2×2 table with cell configurations $\{(1,3),(3,1)\}$ for the top left cell, and $\{(2,2)\}$ for the remaining cells, with a width limit of 5. The minimal height of the first row is 2. The minimal width of the first column is 2. Hence none of the configurations of the top left cell are both greater than the minimal height and minimal width. The possible column widths for column 1 are then computed as $\{2\}$. The only layout is then choosing configuration $(1,3)$ for the top left cell, giving a total height of 5. Choosing the other configuration leads to a total table height of 4.*

We can fix this model by including in the possible column widths the smallest width configuration for each cell in that column which is less than the minimum row height. For the example above this means the possible columns widths become $\{2,3\}$. Bilauca and Healy [5] give an OPL model of their "improved model", to contrast it with the CP model above we give a corresponding corrected Zinc model MIP.

```
int: m; % number of rows
int: n; % number of columns
int: W; % maximal width
array[1..m,1..n] of set of tuple(int,int): cf;

array[1..n] of int: minW = [ max(r in 1..m)
  (min(t in cf[r,c])(t.1)) | c in 1..n ];
array[1..m] of int: minH = [ max(c in 1..n)
  (min(t in cf[r,c])(t.2)) | r in 1..m ];
```

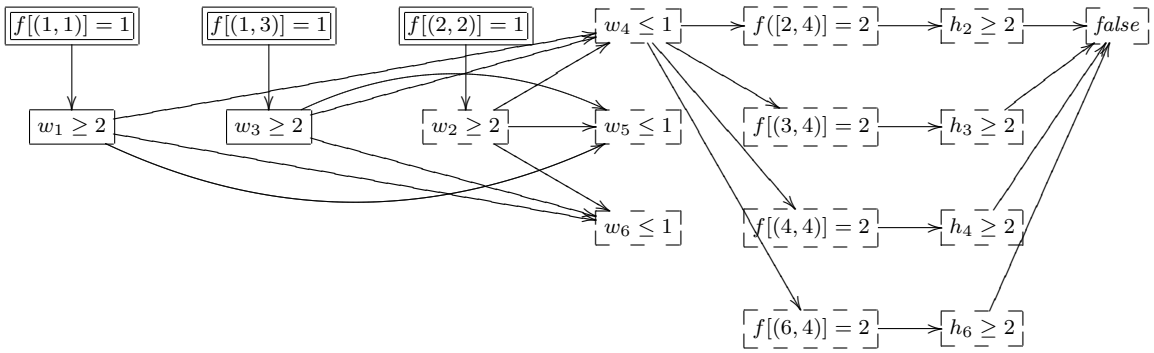

Figure 3: An implication graph for searching the layout problem of Example 4.1

```
array[1..n] of set of int: colWset =
  [ { t.1 | r in 1..m, t in cf[r,c] where
    t.1 >= minW[c] /\ (t.2 >= minH[r] \/
    t.1 == min({ u.1 | u in cf[r,c] %FIX
                where u.2 < minH[r] })) }
  | c in 1..n];

array[1..n] of array[int] of var 0..1: colSel =
  [ [ d:_ | d in colWset[c] ] | c in 1..n ];
array[1..n,1..m] of array[int,int] of var 0..1:
  cellSel =
  array2d(1..n,1..m,[ [ t:_ | t in cf[r,c] ]
                    | r in 1..m, c in 1..n ]);
array[1..n] of var int: w;
array[1..m] of var int: h;

constraint forall(r in 1..m, c in 1..n)(
  sum(t in cf[r,c])(cellSel[r,c][t]) = 1 /\
  sum(t in cf[r,c])(cellSel[r,c][t] * t.1) <= w[c] /\
  sum(t in cf[r,c])(cellSel[r,c][t] * t.2) <= h[r]);
constraint forall(c in 1..n)(
  sum(d in colWset[c])(colSel[c][d]) = 1 /\
  w[c] = sum(d in colWset[c])(colSel[c][d] * d));

constraint sum(c in 1..n)(w[c]) <= W;
solve minimize sum(r in 1..m)(h[r]);
```

Note that Bilauca and Healy also consider a CP model of the problem, but this is effectively equivalent to the MIP model because they do not make use of variable array indices (`element` constraints) that FD solvers support and which allow stronger propagation than that of the 01 encoding.

6. EVALUATION

We compare different approaches to optimal table layout: the A* algorithm of Section 3; the two constraint programming models of Section 4, namely the basic CP implementation without learning (CP-W) and the model using lazy clause generation to provide learning (CP); and BMIP the basic MIP model of [5], and MIP the (corrected) improved model of [5] described in Section 5. For the CP approaches, both CP-W and CP_{seq} use a sequential search that chooses a cell that has the smallest height configuration remaining of all unfixed cells and tries to set it to that minimal height. For the lazy clause solver CP_{vsids} uses the default activity based search.

The $A^\star$ algorithm is written in the high-level declarative programming language Mercury. Notes in Section 3 give some idea of what optimizations have or haven't been applied to the source code of the implementation shown in these timings.

For the constraint programming approaches we used the Chuffed lazy clause generation solver (which can also be run without nogood generation). Chuffed is a state-of-the-art CP solver, which scored the most points in all categories of the 2010 MiniZinc Challenge [14] which compares CP solvers. Since Chuffed does not currently support Zinc, we created the model using the C++ modelling capabilities of Chuffed. The resulting constraints are identical to that shown in the model.

For the MIP approach, we used a script to construct a mixed integer programming model for each table, identical to that created by the Zinc model (and the (corrected) original OPL model of Bilauca and Healy), which was solved using CPLEX 12.1.

We first evaluated the various approaches using a large corpus of real-world tables. This was obtained by collecting more than 10,000 web pages using a web crawler, extracting non-nested tables (since we have not considered how to handle nested tables efficiently), 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. Some of the resulting table layout problems are trivial to find solutions for; we retained only those problems that our $A^\star$ implementation took at least a certain amount of time to solve. (Thus, the choice may be slightly biased against our $A^\star$ solver, insofar as some other solver might find different examples to be hard.) This left 2063 table layout problems in the corpus. We split the corpus into **web-compound** and **web-simple** based on whether the table contained compound cells or not.

Table 1 shows the results of the different methods on the **web-simple** examples. The table shows the number of tables laid out optimally for various time limits up to 10 seconds. They show that in practice for simple tables all of the methods are very good, and able to optimally layout almost all tables very quickly. The worst method is CP-W and the evaluation clearly shows the advantage of learning for constraint programming. We find, like [5], that the improved MIP model MIP while initially slower is more robust than basic model BMIP. Overall CP_{seq} is the most robust approach never requiring more than 1 second on any example. However, the performance of the A* model is surprisingly good

time (s)	CP-W	CP$_{seq}$	CP$_{vsids}$	BMIP	MIP	A*
≤ 0.01	1018	**1097**	1043	1009	774	972
≤ 0.10	1064	**1252**	1186	1160	1076	1168
≤ 1.00	1103	**1271**	1257	1221	1223	1262
≤ 10.00	1120	**1271**	1269	1261	1270	1269
> 10.00	151	**0**	2	10	1	2

Table 1: Number of instances from the web-simple data-set solved within each time limit.

time (s)	CP-W	CP$_{seq}$	CP$_{vsids}$	BMIP	A*
≤ 0.01	708	**713**	665	702	636
≤ 0.10	721	**774**	737	760	751
≤ 1.00	734	**788**	771	787	787
≤ 10.00	742	790	778	790	**792**
> 10.00	50	2	14	2	**0**

Table 2: Number of instances from the web-compound data-set solved within each time limit.

given the relative simplicity of the approach in comparison to the sophisticated CPlex and Chuffed implementations and the use of Mercury rather than C or C++.

Table 2 shows the results of the different methods on the **web-compound** examples. We compare all the previous algorithms except for MIP since it is not applicable when there are compound cells. The results are similar to those for simple tables. For these more complicated tables, the $A^\star$ approach is slightly more robust, while CP$_{seq}$ is the fastest for the easier tables.

Given the relatively similar performance of the approaches on the real-world tables we decided to "stress-test" the approaches on some harder artificially constructed examples. We only used simple tables so that we could compare with MIP. Table 3 shows the results. We created tables of size $m \times n$ each with k configurations by taking text from the Gutenberg project edition of The Trial [10] k words at a time, and assigning to a cell all the layouts for that k words using fixed width fonts. For the experiments we used $k = 6$. We compare different versions of the layout problem by computing the minimum width $minw$ of the table as the sum of the minimal column widths, and the maximal width $maxw$ of the table as the sum of the column widths resulting when we choose the minimal height for each row. The *squeeze* s for table is defines as $(W - minw)/maxW$. We compare the table layout for 5 different values of squeeze. Obviously with a squeeze of 0.0 or 1.0 the problem is easy, the interesting cases are in the middle.

The harder artificial tables illustrate the advantages of the conflict directed search of CP$_{vsids}$. On the **web-simple** and **web-compound** corpora, the approaches with more naive search strategies, CP$_{seq}$ and A*, performed best. However, they were unable to solve many of the harder tables from the **artificial** corpus, where CP$_{vsids}$ solved all but one instance, sometimes 2 orders of magnitude faster than any other solver. Interestingly the MIP approach wins for one hard example, illustrating that the automatic MIP search in CPLEX can also be competitive.

The difference in behavior between the real-world and artificial tables may be due to differences in the table structure. The tables in the **web-simple** and **web-compound** corpora tend to be narrow and tall, with very few configurations per cell – the widest table has 27 columns, compared

	s	CP$_{seq}$	CP$_{vsids}$	BMIP	MIP	A*
10×10	0.00	**0.00**	**0.00**	**0.00**	**0.00**	0.02
	0.25	3.29	**0.02**	0.16	0.97	0.06
	0.50	0.3	**0.01**	0.22	0.56	0.03
	0.75	0.07	**0.02**	0.55	1.18	0.09
	1.00	**0.00**	**0.00**	0.01	0.01	**0.00**
20×20	0.00	0.02	**0.01**	**0.01**	0.04	0.19
	0.25	—	**0.86**	27.18	28.65	37.04
	0.50	—	**0.07**	188.27	163.86	11.44
	0.75	—	**0.28**	43.83	40.07	62.03
	1.00	0.02	0.01	0.04	0.08	**0.00**
30×30	0.00	0.04	**0.03**	0.04	0.07	1.53
	0.25	—	254.47	—	**253.08**	—
	0.50	—	**0.38**	—	—	—
	0.75	—	**9.4**	—	—	—
	1.00	0.04	0.04	0.10	0.18	**0.00**
40×40	0.00	0.09	**0.06**	0.07	0.20	3.55
	0.25	—	—	—	—	—
	0.50	—	**1.11**	—	—	—
	0.75	—	**216.67**	—	—	—
	1.00	0.09	0.05	0.19	0.34	**0.02**

Table 3: Results for artificially constructed tables. Times are in seconds.

with 589 rows, and many cells have only one configuration. On these tables, the greedy approach of picking the widest (and shortest) configuration tends to quickly eliminate tall layouts. The **artificial** corpus, having more columns and more configurations, requires significantly more search to prove optimality; in these cases, the learning and conflict-directed search of CP$_{vsids}$ provides a significant advantage.

7. CONCLUSION

Treating table layout as a constrained optimization problem allows us to use powerful generic approaches to combinatorial optimization to tackle these problems. We have given three new three new techniques for finding a minimal height table layout for a fixed width: the first uses an $A^\star$ based approach while the second approach uses pure constraint programming (CP) and the third uses lazy clause generation, a hybrid CP/SAT approach. We have compared these with two MIP models previously proposed by Bilauca and Healy.

An empirical evaluation against the most challenging of over 50,000 HTML tables collected from the Web showed that all methods can produce optimal layout quickly.

The $A^\star$-based algorithm is more targeted than the constraint-programming approaches: while the $A^\star$ algorithm did well on the web-page-like tables for which it was designed, we would expect that more generic constraint-programming approaches would be a safer choice for other types of large tables. This turned out to be the case for the large artificially constructed tables we tested, where the approach using lazy clause generation was significantly more effective than the other approaches.

All approaches can be easily extended to handle constraints on table widths such as enforcing a fixed size or that two columns must have the same width. Handling nested tables, especially in the case cell size depends in a non-trivial way on the size of tables inside it (for example when floats are involved) is more difficult, and is something we plan to pursue.

8. ACKNOWLEDGEMENTS

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

9. REFERENCES

[1] R. J. Anderson and S. Sobti. The table layout problem. In *SCG '99: Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, pages 115–123, New York, NY, USA, 1999. ACM Press.

[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] R. J. Beach. *Setting tables and illustrations with style*. PhD thesis, University of Waterloo, 1985.

[4] N. Beaumont. Fitting a table to a page using non-linear optimization. *Asia-Pacific Journal of Operational Research*, 21(2):259–270, 2004.

[5] M. Bilauca and P. Healy. A new model for automated table layout. In *Proceedings of the 10th ACM symposium on Document engineering*, DocEng '10, pages 169–176. ACM, 2010.

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

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

[8] N. Hurst, K. Marriott, and D. Albrecht. Solving the simple continuous table layout problem. In *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, pages 28–30, New York, NY, USA, 2006. ACM.

[9] N. Hurst, K. Marriott, and P. Moulder. Towards tighter tables. In *Proceedings of Document Engineering, 2005*, pages 74–83, New York, 2005. ACM.

[10] F. Kafka. *The Trial*. Project Gutenberg, 1925, 2005.

[11] X. Lin. Active layout engine: Algorithms and applications in variable data printing. *Computer-Aided Design*, 38(5):444–456, 2006.

[12] K. Marriott, N. Nethercote, R. Rafeh, P. Stuckey, M. Garcia de la Banda, and M. Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

[13] K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, Massachusetts, 1998.

[14] 2010 MiniZinc challenge. `http://www.g12.csse.unimelb.edu.au/minizinc/challenge2010/results2010.html`, 2010.

[15] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, 2001.

[16] O. Ohrimenko, P. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[17] D. Raggett, A. L. Hors, and I. Jacobs. HTML 4.01 Specification, section 'Autolayout Algorithm'. http://www.w3.org/TR/html4/appendix/notes.html#h-B.5.2, 1999.

[18] S. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach*. Prentice Hall, 2nd edition, 2002.

[19] X. Wang and D. Wood. Tabular formatting problems. In *PODP '96: Proceedings of the Third International Workshop on Principles of Document Processing*, pages 171–181, London, UK, 1997. Springer-Verlag.