Toward Tighter Tables

Nathan Hurst, Kim Marriott, and Peter Moulder
Clayton School of Information Technology, Monash University
Clayton, Victoria, Australia
{njh,marriott,pmoulder}@infotech.monash.edu.au

ABSTRACT

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. Unfortunately, optimal layout of tables which contain text is NP-hard for reasonable layout requirements such as minimizing table height for a given width [1]. We present two new independently-applicable techniques for table layout. The first technique is to solve a continuous approximation to the original layout problem by using a constant-area approximation of the cell content combined with a minimum width and height for the cell. The second technique starts by setting each column to its narrowest possible width and then iteratively reduces the height of the table by judiciously widening its columns. This second technique uses the actual text and line-break rules rather than the constant-area approximation used by the first technique. We also investigate two hybrid approaches both of which use iterative column widening to improve the quality of an initial solution found using a different technique. In the first hybrid approach we use the continuous approximation technique to compute the initial column widths while in the second hybrid approach a modification of the HTML table layout algorithm is used to compute the initial widths. We found that all four techniques are reasonably fast and give significantly more compact layout than that of HTML layout engines.

Categories and Subject Descriptors

I.7.2 [**Document Preparation**]: Format and notation—*automatic table layout,heuristics,optimisation*

General Terms

Algorithms

Keywords

table layout, optimisation techniques, conic programming

DocEng'05, November 2–4, 2005, Bristol, United Kingdom.

1. INTRODUCTION

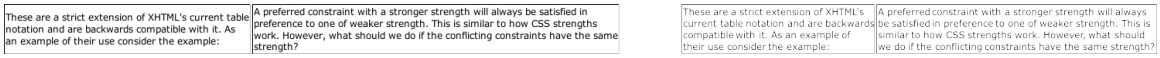
Tables are provided in virtually all document formatting systems and are one of the most powerful and useful design elements in current web document standards such as (X)HTML, CSS and XSL. Indeed because of their power, tables are frequently (mis)used by web designers to finely control page layout, not just to display tabular information.

Unfortunately automatic layout of tables which contain text is not an easy task. Both Wang and Wood [11] and Anderson and Sobti [1] have proven that table layout with text is NP-hard. The underlying 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 not too surprising that it is NP-hard to find which combination of these discrete configurations best satisfies reasonable layout requirements such as minimizing table height for a given width.

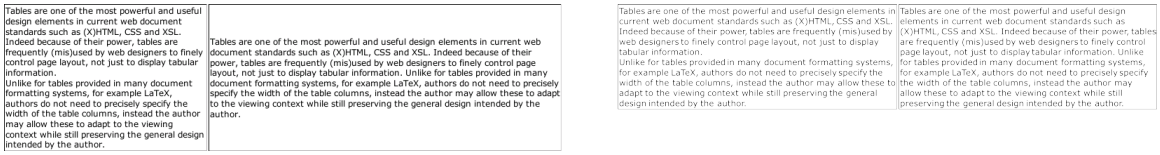
For this reason most document formatting systems require the author to control column width in tables containing text, something that turns table formatting into a tedious chore. However, in online documents it is not practical to require the author to exactly fix table column widths at document authoring time since the table layout needs to adjust to different width viewing environments and to different text sizes since, for instance, the viewer may choose a larger font, or some of the text may be generated dynamically. Thus in the current (X)HTML, CSS and XSL table specification, authors do not need to precisely specify width of table columns, instead the author may allow these to adapt to the viewing context while still preserving the general design intended by the author.

Unfortunately, standard automatic table layout engines for HTML, CSS and XSL can give poor layout for tables containing significant amounts of text or multi-column or multi-row cells. Here we present four new approaches to table layout all of which are reasonably fast and give better layout than current HTML engines. As an example consider the three tables shown in Figure 1 which compare table layout using Mozilla against the layout we obtain with one of our approaches.

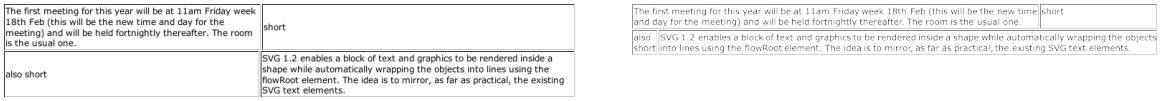
The key insight behind our first approach is that at least for cells with more than a few words of text the cell configurations can be approximated by a continuous (non-linear) constraint that the cell is large enough to contain the area of its contents. As some evidence for this claim we have plotted the area of the paragraph divided by the text area against paragraph width for all minimal text configurations for the complete works of the Australian poet, C.J. Dennis (some 154 paragraphs with about 20,000 words.) Note that the text area is the area of the text when laid out in a single line. The results are shown in Figure 2. It is clear that area approximation is accurate except when the cell is very narrow.



Example *2n2-linear* is a very simple table of two cells, demonstrating that our methods can give more compact layout even for very simple tables



Example *multipara* has two cells with identical text except that one of them contains line breaks and demonstrates the HTML algorithm's poor handling of multi-paragraph text



Example *simple-brick* demonstrates Mozilla's poor handling of compound cells. Note that the table has 3 columns but because the middle column has no non-compound cells, Mozilla assigns it zero width as is allowed by the HTML algorithm

Figure 1: Three example tables comparing layout using Mozilla (on the left) with our first hybrid approach (on the right).

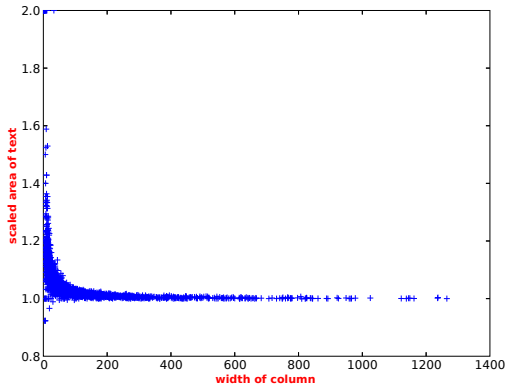


Figure 2: Plot of actual paragraph area divided by text area as a function of paragraph width for the complete works of the Australian poet, C.J. Dennis.

This suggests an approach in which we first solve a continuous approximation to the original layout problem which has minimum values for the column widths and row heights reflecting minimum cell widths and text height but rather than requiring each cell to be big enough to contain its contents we instead require that it is large enough to contain the area of its contents. We then find a solution to the original problem by setting the column widths to their value in the optimal solution to the continuous table layout problem and laying out the text in each cell and setting the row heights to the minimum value respecting the constraints. The advantage of approximating the table layout problem by a continuous constrained optimisation problem is that this is easier to solve than the original NP-hard problem, and, as we show, can be solved in polynomial time using conic programming. We call this approach *area approximation*.

Our second approach, which we call *iterative column widening*, starts by setting each column to its narrowest possible value by taking into account minimum widths for each of the cells in the table and then iteratively reducing the height of a row (and hence the table) by judiciously widening table columns. Efficient implementation of iterative column widening requires efficient computation of the next shortest cell configuration, i.e. the minimum width which allows the text in a cell to be laid out in one less line. We give a simple, linear time dynamic programming algorithm to compute this.

Our third approach combines the previous two approaches in a hybrid algorithm in which we first solve the continuous approximation to the original layout problem and then, in a second phase, use iterative column widening to improve this solution.

Our fourth approach is a variant of this hybrid approach in which we use a modification to HTML's 'Autolayout Algorithm' [9] to compute the initial column widths and then use iterative column widening.

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

Another application of our work is for better, more powerful table layout in document formatting systems such as LaTeX and word processors. Also our techniques can be used to improve formatting of spreadsheets for printing on fixed size pages. Existing tools do quite badly, often resulting in a single column being printed on an extra page. The techniques presented here can be used to efficiently choose between portrait and landscape, and to determine the minimum number of pages to use.

This is not the first paper to model table layout as constrained optimisation. Beach's thesis [4] presented the table layout problem formally. Wang and Wood [11] investigated the problem of

semantic modelling of tables and presented a branch and bound algorithm, accelerated with a polynomial-time greedy algorithm. The iterative widening algorithm is related to Wang and Wood's [11] greedy algorithm. One difference is that we are interested in minimising an objective function (the table height) rather than just finding the first solution which satisfies the constraints. The second difference is that our algorithm is a heuristic, it never uses a branch and bound search.

Anderson and Sobti [1] provide further analysis of the complexity of the table layout problem and give an efficient algorithm for generating all minimal configurations for a text cell. They investigate a linear programming approximation to the problem in which the convex hull of the configurations is modelled as a conjunction of linear constraints. This is related to the area approximation algorithm. Their approach has the disadvantage that all minimal configurations for each text cell must be computed and then their convex hull. Since the number of minimal text configurations can be linear in the number of words in the cell this is expensive and can lead to a linear program which has a linear number of constraints in the number of words in the table.

Other related work includes that of Borning et al [5, 6, 3] who allowed the designer to specify required and preferred linear arithmetic constraints over column widths. A linear constraint solver is then used to determine column widths.

2. THE TABLE LAYOUT PROBLEM

We first formalise the *table layout problem*. We assume throughout this paper that the table of interest has n columns and m rows and that w_c is the width of column c and h_r the height of row r. The designer must specify how the grid elements of the table are partitioned into logical elements or cells. We call this the *table structure*. We distinguish between simple and compound cells. A simple cell is a single grid element (i.e. it 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 set of rows spanned by d and $cols(d)$ to be the set of columns spanned by d. We let $bottom(d) = \max rows(d)$ and $right(d) = \max cols(d)$. Each cell d has a minimum width, $minwidth(d)$, which is the length of the longest word in the cell, a minimum height, $minheight(d)$, which is the height of the highest word, and a line width $linewidth(d)$ which is the length of the cell contents when laid out in a single line.

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

- *fixed size* for selected column widths[1]
- *lower bounds* on selected column widths or row heights
- *fixed ratios* between selected column widths

The final aspect to table layout is the *layout style*. This captures what is required in a good layout and is generic, capturing a particular kind of style objective. We shall focus on the *minimum height* layout style: Given a page width find a layout for the table that, in decreasing order of importance, is no wider than the page width, minimizes table height, and minimizes table width.

The table layout problem is, given a table structure and content for the table cells, some designer constraints and a layout style, to find an assignment to the column widths and row heights s.t.: (a) the cells are large enough to contain their content; (b) the designer constraints are satisfied; and (c) the layout style is satisfied. In the case of conflict, these are in order of decreasing importance.

The main reason for this choice of features is that it generalises tables provided in standard document processing software including HTML.

HTML 4.0, CSS 2 and XSL provide similar table formalisms. The designer can specify the table as a collection of simple and compound rectangular cells. They can specify that a column has an absolute width, such as 200 pixels; or a percentage width relative to the width of the table (in CSS2 and XSL) or of the object surrounding the table (in HTML). They can also specify the width of the table in this way. HTML allows the designer to specify that a column should have width that is some fixed ratio of the special width "$*$".

The suggested layout algorithm for HTML tables is quite complex [9] but essentially works as follows. The effect is to try and minimize table height for a particular width. The minimum width for a column is just the maximum of the minimum widths of the cells in the column while the line width is the maximum of the line widths of the cells in the column. For multi-column cells the line width is split between the columns, exactly how is left to the implementation. Fixed width columns are given the size specified by the user. The widths of the unfixed width columns are scaled to fit the remaining width. The scaling for each column is proportional to the difference between the column's line width and minimum width : thus the line width is regarded as the column's desired width but no column will have its width scaled below its minimum width. Once the column widths are computed the cell contents are placed in the cells and the minimum height of each cell computed. A greedy text layout algorithm is used to place text. The row height is set to the maximum of the row's cell heights.

As we have seen, this approach can give quite bad layout if a cell contains more than one paragraph, or the table has multi-column or multi-row cells.

3. AREA APPROXIMATION

The main idea behind our first approach to table layout is that for cells with more than a few words of text the cell configurations can be approximated by a continuous (non-linear) constraint that the cell is large enough to contain the area of its contents. We solve a continuous approximation to the original layout problem which has the same designer constraints and layout style as the original problem but rather than requiring a cell to be big enough to contain its textual content we instead require that for each cell d

$$area(d) \leq (\sum_{r \in rows(d)} h_r) \times (\sum_{c \in cols(d)} w_c) \quad (1)$$

where $area(d)$ is the area of the contents of d. In the case of text this includes the area of the words and a separator between adjacent words. We also add linear constraints to ensure that a cell is wider than its longest word and taller than its highest word. We call such a problem the *continuous table layout problem*.

We solve the continuous layout problem to determine the column widths in the original problem and then layout the text in each cell to determine the minimum height of the rows. This is similar to how HTML layout is performed once column widths have been determined.

A constrained optimisation problem is said to be a *convex programming problem* if the objective function is convex and the solutions to the constraints form a convex set, i.e. any point between two solutions is also a solution. Convex programming problems have the important property that any locally optimal solution is a

[1]Note that specifying that a table column is $x\%$ of the page width is a fixed size constraint

globally optimal solution [7], which makes them considerably easier to solve than non-convex problems. Fortunately:

LEMMA 1. *The continuous table layout problem is a convex programming problem*

This follows because we can rewrite the problem into a linear program plus a conjunction of simple area constraints of the form $c \leq x \times y$ where x and y are constrained to be the sum of the column widths and row heights respectively. Now such constraints are convex as are linear constraints, so the conjunction of designer constraints is convex. Furthermore the minimum height layout style can be modelled by a linear objective function and so the entire problem is a convex programming problem.

This result suggests that we can use general purpose mathematical optimisation techniques for solving convex programming problems. Unfortunately, neither linear programming nor quadratic programming techniques are powerful enough, but we can use a more powerful technique known as *conic programming*. Conic programming is a generalisation of quadratic programming that allows certain curved constraints. Conic programs can be solved in polynomial time using recently developed interior point methods. These methods have reached sufficient maturity to provide a practical, efficient approach for solving such problems. For more details we refer the reader to [10].

We have used the Mosek conic programming optimisation kit [2]. This solves problems of the following form: The first part of the problem is a standard linear program over the variables $x_1, \ldots, x_n$. This consists of a linear objective function to be minimised, an optional lower and upper bound for each variable x_i and a conjunction of linear inequality constraints over the x_i each of which has form: $l \leq \sum_{i=1}^{n} a_i x_i \leq u$. Observe that if the lower bound l and upperbound u are equal then this is an equality. The second part of a conic program is a conjunction of "cone" constraints each of which is over a different set of variables $y_1, \ldots, y_{n_c}$ a subset of $x_1, \ldots, x_n$. A cone has one of two forms:

$$y_1 \geq \sqrt{\sum_{j=2}^{n_c} y_j^2} \qquad \text{or} \qquad 2 \cdot y_1 \cdot y_2 \geq \sum_{j=3}^{n_c} y_j^2.$$

A variable can appear in at most one cone constraint, but they can be effectively placed in multiple cones by creating mirror variables and equating these in the linear constraints.

It is relatively simple to model the continuous table layout problem as a conic program of the above form. The basic area constraint for cell d given in Equation 1 can be modelled using the second form of cone over three variables x_1, x_2, x_3:

$$2 \cdot x_1 \cdot x_2 \geq \sum_{j=3}^{3} x_3^2$$

with linear constraints equating

$$x_1 = \sum_{r \in rows(d)} h_r \qquad \text{and} \qquad x_2 = \sum_{c \in cols(d)} w_c$$

and setting both the lower and upper bound of x_3 to $\sqrt{2 \cdot area(d)}$.

Since arbitrary linear constraints over the row and column variables are allowed in the conic program it is simple to model designer constraints such as fixed size, minimum width or height, or fixed ratios between column widths. Modelling the minimum height layout style is also simple: The objective function to be minimised is $\sum_{r=1}^{m} h_r$ and we add the constraint

$$0 \leq \sum_{c=1}^{n} w_c \leq \text{table width}$$

to ensure the table is no wider than the maximum allowed width.

A simple example is a 2x2 table with two single cells in the first row and a 2-column compound cell on the second row:

	w_1	w_2
h_1	A_1	A_2
h_2	A_3	

We have three constraints we need satisfied:

$$h_1 \cdot w_1 \geq A_1, \qquad h_1 \cdot w_2 \geq A_2, \qquad h_2 \cdot (w_1 + w_2) \geq A_3$$

The first step is to transform the compound cells into simple cells using a linear transformation. We construct a variable $w = w_1 + w_2$ for the compound cell and a variable for each area term. We also need a separate variable for each extra use of a row or column, in this case we introduce $h_{1,1}$ and $h_{1,2}$ as aliases for h_1. Finally, we construct the special "constant" variables a_i which provides the area constraint bound. The complete set of variables and bounds is

$$\begin{gathered} 0 \leq h_1, \qquad 0 \leq h_{1,1}, \qquad 0 \leq h_{1,2}, \qquad 0 \leq h_2, \\ 0 \leq w_1, \qquad 0 \leq w_2, \qquad 0 \leq w, \\ \sqrt{2 \cdot A_1} \leq a_1 \leq \sqrt{2 \cdot A_1}, \\ \sqrt{2 \cdot A_2} \leq a_2 \leq \sqrt{2 \cdot A_2}, \\ \sqrt{2 \cdot A_3} \leq a_3 \leq \sqrt{2 \cdot A_3} \end{gathered}$$

(In our code we actually create a separate variable for each row and column variable each time we use them, but this clutters things.)

We have 3 cones, one for each cell:

$$2 \cdot h_{1,1} \cdot w_1 \geq a_1^2, \quad 2 \cdot h_{1,2} \cdot w_2 \geq a_2^2, \quad 2 \cdot h_2 \cdot w \geq a_3^2$$

which are linked using the linear constraints:

$$\begin{array}{rcccl} 0 & \leq & h_1 - h_{1,1} & \leq & 0 \\ 0 & \leq & h_1 - h_{1,2} & \leq & 0 \\ 0 & \leq & w - w_1 - w_2 & \leq & 0 \\ 0 & \leq & w_1 + w_2 & \leq & \text{table width} \end{array}$$

And our objective function to be minimised is $h_1 + h_2$.

One of the main strengths of using conic programming to solve the continuous table approximation problem is that, because it allows arbitrary linear constraints, it is straightforward to handle more complex designer constraints such as fixed aspect ratios for cells, maximum widths on cells or columns, and fixed ratios between cell height and widths (including compound cells) rather than just between columns and rows. It is also simple to handle different layout styles, such as minimizing table width for a given height, or finding the most compact layout for a given aspect ratio since these can be couched as minimising a linear objective function. Furthermore, embedded tables are naturally treated as just another kind of cell with a minimum area, width and height constraint. One extension that cannot be handled is to allow non-rectangular compound cells: this leads to a non-convex problem, which cannot be modelled with conic programming alone.

4. ITERATIVE COLUMN WIDENING

Our second approach to table layout starts by setting each column to its narrowest possible value by taking into account minimum widths for each of the cells in the table and then iteratively reducing the height of a row and hence the table by judiciously widening table columns. The detailed algorithm is shown in Figure 3. For simplicity we first ignore designer constraints.

The algorithm relies on two functions for computing information about minimal text configurations for cells. The first function, $min\_height(d, \phi, \psi)$, returns the minimum height for the row $bottom(d)$ that allows the contents of cell d to fit given width $\phi(w_c)$

column-widening(*max_width*)
ϕ := *minimum-col-widths*()
ψ := *compute-row-heights*(ϕ)
(ϕ,ψ) = *reduce-table-height*(ϕ,ψ, *max_width*)
return (ϕ, ψ)

minimum-col-widths()
for each column $c = 1, ..., n$ **do**
  $cells\_c := \{d \mid d \text{ a cell s.t. } right(d) = c\}$
  $min\_width := \max\{0\} \cup$
    $\{minwidth(d) - \sum_{c \in cols(d) \setminus \{right(d)\}} \phi(w_c) \mid d \in cells\_c\}$
  $\phi := \phi[w_c \mapsto min\_width]$
end for
return ϕ

compute-col-widths(ψ)
$\phi := [\,]$
for each column $c = 1, ..., n$ **do**
  $cells\_c := \{d \mid d \text{ a cell s.t. } right(d) = c\}$
  $min\_width := \max\{0\} \cup \{min\_width(d, \phi, \psi) \mid d \in cells\_c\}$
  $\phi := \phi[w_c \mapsto min\_width]$
end for
return ϕ

reduce-table-height(ϕ,ψ, *max_width*)
repeat
  $current\_width := \sum_{c=1}^{n} \phi(w_c)$
  $current\_height := \sum_{r=1}^{m} \psi(h_r)$
  $best := 0$
  $best\_score := 0$
  for each row $r = 1, ..., m$ **do**
    $cells\_r := \{d \mid d \text{ a cell s.t. } bottom(d) = r\}$
    $min\_height\_r := \max\{0\} \cup$
      $\{minheight(d) - \sum_{r' \in rows(d) \setminus \{bottom(d)\}} \psi(h_{r'}) \mid d \in cells\_r\}$
    if $min\_height\_r \leq \psi(h_r) - \varepsilon$ **then**
      $\psi' := \psi[h_r \mapsto \psi(h_r) - \varepsilon]$
      $\phi' :=$ *compute-col-widths*(ψ')
      $\psi' :=$ *compute-row-heights*(ϕ')
      $new\_width := \sum_{c=1}^{n} \phi'(w_c)$
      $new\_height := \sum_{r=1}^{m} \psi'(h_r)$
      $score := \frac{current\_height - new\_height}{new\_width - current\_width + 1}$
      if $score > best\_score$ **and** $new\_width \leq max\_width$ **then**
        $best := r$
        $best\_score :=$ score
      end if
    end if
  end for
  if $best = 0$ **then return** (ϕ, ψ) **end if**
  $\psi := \psi[h_{best} \mapsto \psi(h_{best}) - \varepsilon]$
  $\phi :=$ *compute-col-widths*(ψ)
  $\psi :=$ *compute-row-heights*(ϕ)
forever

Figure 3: Iterative column widening table layout algorithm

for all columns $c \in cols(d)$ and height $\psi(h_r)$ for the other rows $r \in rows(d) \setminus \{bottom(d)\}$. The second function $min\_width(d, \phi, \psi)$ is dual to *min_height*. It returns the minimum width for the column $right(d)$ that allows the contents of cell d to fit given height $\psi(h_r)$ for the rows $r \in rows(d)$ and widths $\phi(w_c)$ for the other columns $c \in cols(d) \setminus \{right(d)\}$. Efficient implementation of *min_width* and *min_height* is important for the efficiency of this approach and is discussed in Section 6.

Using *min_width* it is simple to define *compute-col-widths* which takes an assignment to the row heights ψ and computes the minimum width for each column which allows the cell contents to fit in each column for the given row heights. The only subtlety is the treatment of compound cells: they are only considered when the rightmost column in the compound cell is reached. Dually, *compute-row-heights* can be defined in terms of *min_height*. For brevity we do not include the code for *compute-row-heights*.

The function *minimum-col-widths* is similar to *compute-col-widths* but instead of using *min_width* to compute the minimum width of the cell for a particular row height assignment it uses the minimum allowed width for each cell in order to determine the minimum width for each column.

The main function *column-widening*(*max_width*) takes a table and the maximum desired width for the table. The algorithm starts by computing the narrowest possible width for each column and then, given these widths, the minimum height for each row in the table. The core part of layout is performed by the function *reduce-table-height*. This iteratively improves the current configuration for the table by choosing a row and reducing the height of that row by widening some columns. This is repeated until it is no longer possible to reduce the height of any row without violating the maximum width for the table. At each step we use a heuristic to choose which row should have its height decreased. The for-loop inside the main loop does this. It uses *compute-col-widths* followed by *compute-row-heights* to compute for each row j a new configuration (ϕ', ψ') for the table in which row j's height is strictly less by at least ε and the height of no other row increases. Note that this is not always possible since the minimum height for a row may have been reached. If the new configuration is valid, i.e. not too wide, a score for its utility is computed. We simply compute the ratio of height reduction to increased table width but more complex scoring mechanisms are also possible. The row with the greatest score is chosen for height reduction.

It is reasonably straightforward to extend the algorithm to handle designer constraints (although we have not yet done so.) For ratio constraints we can collapse columns constrained to be in a fixed ratio to a single column. The only trick is that such columns now have a multiplier for each cell indicating their effective width or height.

In the case of minimum width or height constraints we simply modify *compute-row-heights* and *compute-col-widths* so that they never set a row or column (respectively) to a value less than the minimum.

Fixed width columns are a little more difficult. The first issue is that we do not want a column with fixed size to be at the right of a compound cell since this will mean that we cannot expand the cell properly. Instead we use an ordering on columns in which the first columns are those of fixed width and the term "right" refers to the maximum column in the cell with respect to this new ordering. We must then modify *compute-col-widths* so that it always sets a fixed width column to its desired width.

We believe it is possible to extend the iterative column widening algorithm to handle other designer constraints and table design styles, but at the risk of making the algorithm more complex and

somewhat ad hoc. Unlike conic programming, iterative column widening does appear to generalise to non-rectangular compound cells. We plan to investigate this further. It also appears reasonably simple to handle embedded tables: they behave just like text cells having a set of minimal configurations which the algorithm iterates through.

We also note that the algorithm definition given here is potentially quite inefficient since at each step we may recompute many things, in particular cell configurations. Our actual implementation uses caching to improve efficiency.

5. TWO HYBRID ALGORITHMS

Iterative column widening is actually a general heuristic for improving table layout given any initial width for the columns. Thus is can be used as second phase with say the HTML layout algorithm or with the Area Approximation Algorithm. Importantly, it is an *any time* optimisation algorithm–it can be stopped at any point in time and the current configuration will be a valid table layout which is better than the initial layout, although it may be less than optimal. We have therefore investigated two hybrid approaches to table layout.

In the first hybrid approach we use the Area Approximation Algorithm to compute column widths and then improve this with the Column Widening Algorithm. More exactly, the second phase takes the solution θ to the continuous approximation of a table layout problem. We assume that θ makes each cell at least as wide as the cell's minimum width. We first set the column widths ϕ to $[w_i \mapsto \theta(w_i) \mid i = 1,\ldots,n]$ and compute the row heights ψ using *compute-row-heights*. We now narrow the column widths ψ' by using *compute-col-heights* and then call *reduce-table-height* with ϕ and ψ to improve this layout.

An example is shown in Figure 4. The example shows how this second phase reduces the height of the table by two lines.

In our second hybrid approach we use a variant of the HTML algorithm to compute the initial column widths rather than the Area Approximation Algorithm. Our variant uses the function *minimum-col-widths* from Figure 3 to compute the minimum column widths and the obvious modification to this function (in which the call to $minwidth(d)$ is replaced by $linewidth(d)$) to compute the line width for each column. This is in accord with the HTML specification since it has the same behaviour for non multi-column cells but we believe gives better, more principled behaviour for multi-column cells than do standard implementations of the HTML table layout algorithm.

6. EFFICIENT COMPUTATION OF MINIMAL CELL CONFIGURATIONS

Efficient implementation of *min_width* and *min_height* is important for good performance of the Iterative Column Widening Algorithm and the two hybrid algorithms. These compute the narrowest (shortest) minimal configuration for the cell which is less than a given height (width) respectively, where a minimal configuration is a pair (w,h) s.t. the cell contents can be laid out in a rectangle with width w and height h but there is no smaller pair that fits the contents, i.e. for all $w' \leq w$ and $h' \leq h$, either $h = h'$ and $w = w'$, or the cell contents do not fit in a rectangle with width w' and height h'.

In the case of images, there is only one such minimal configuration, so this is easy; but for text with uniform height with n words there are up to n minimal configurations with each corresponding to a different number of lines, while for text with non-uniform height there may be $O(n^2)$ minimal configurations. Currently our implementation only handles cells with text of uniform height but we are extending it to handle non-uniform height text.

Given a fixed width for a rectangular cell it is reasonably simple to compute the minimum height for the cell which will allow the cell's text to fit in. For text with uniform height a quick and simple way is to use a greedy algorithm for text layout in which text is placed sequentially upon each line until it is full. This has linear complexity. [2] In the case of text with non-uniform height, the greedy text layout algorithm will compute the minimum number of lines required but may not compute the minimum height. For computing minimal height for text with non-uniform height, it is possible to use a variant of the dynamic programming algorithm of Knuth–Plass [8] for computing optimal line breaking in paragraphs.

Computing a cell's minimum width for a fixed height is less easy. However in our context we are primarily interested in computing the next shortest configuration. This suggests that we actually need an efficient algorithm that, given a minimal configuration (w,h) for a cell d, will compute the next shorter minimal configuration, i.e. the tallest minimal configuration (w',h') for d s.t. $h' < h$. This allows us to incrementally compute the minimal configurations as needed.

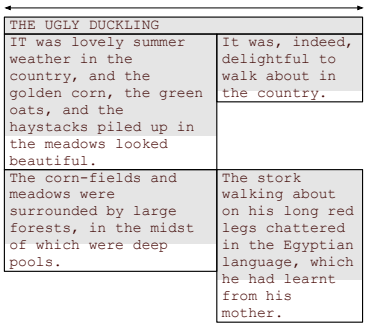
The obvious approach to compute the next shorter minimal configuration is to start with the configuration (w,h) and compute the layout for width w with the greedy algorithm and then determine the minimal increase to the width, say e, that will allow an extra word to fit on some line in the layout. Next increase w to $w+e$ and compute the layout again, continuing this widening process until the number of lines decreases in which case the next shorter minimal configuration has been reached.

However, we have found a more efficient method. Figure 5 gives a function *next-shorter-cell-configuration* for computing the next shorter minimal configuration for a cell in linear time assuming the text has uniform height. Note that for simplicity we assume a mono-spaced font but it is straightforward to handle non mono-spaced fonts. We assume that there are n words and that the array $width[w]$ contains the width of word w. The algorithm starts from the layout computed by the greedy layout algorithm for the current width $cwidth$. We assume that the layout has $lastLine$ lines and that the array $lineStart[L]$ gives the index of the word starting line L. The algorithm uses dynamic programming to compute $minWidth[w]$ which gives the narrowest width required to layout words $w\ldots n$ in less lines than the current configuration assuming that a line now starts at w, i.e., if w is on line L then the minimum width required to layout words $w\ldots n$ in no more than $lastline - L$ lines. Actually if this narrowest width is less than $cwidth$ we do not compute it exactly but rather set $minWidth[w]$ to $cwidth$ since the exact value will be ignored as we know that the final value must be greater than $cwidth$. The algorithm works backwards from the last line. Clearly, $minWidth[1]$ is the width of the next shorter minimal configuration.

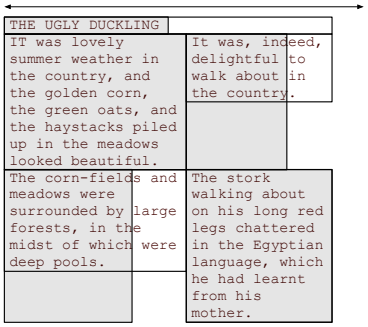
As an example, consider the following text laid out in a width of 15. For simplicity we have used a mono-spaced font. The words are annotated with their index.

```
It[1] was,[2] indeed,[3]
delightful[4] to[5]
```

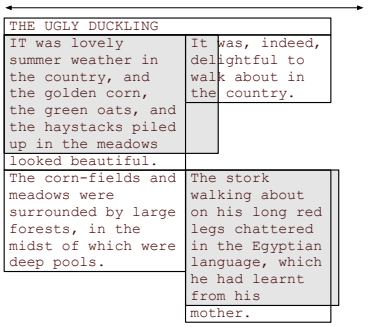
[2] We note that Anderson and Sobti [1] sketch a rather complex algorithm for finding the minimal height for a cell in, as best we understand, $O(n^{1/2})$ time where there are n words although this requires a once-off pre-computation which takes $O(n^{3/2}\log m)$ time where m is the length of the longest word. They use this to compute all minimal configurations in, we believe, $O(n^{3/2}\log N)$ time where n is the number of words and N the text length.



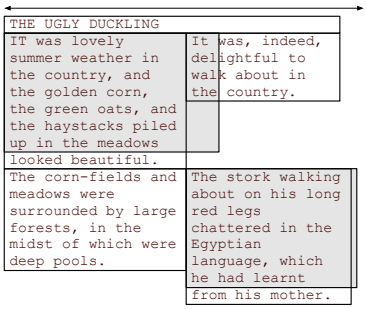
(a) Initial layout resulting from using the Area Approximation Algorithm. The arrow above the table gives the maximum allowed table width and the grey in each cell shows the approximate cell area used in the approximation.



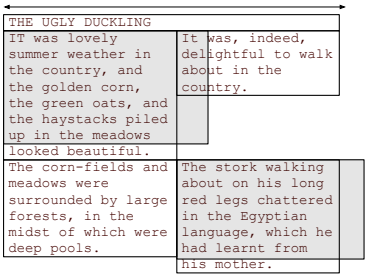
(b) Layout after narrowing each column. The grey box shows the narrowest width for each cell that does not increase the row height.



(c) The algorithm computes for each row the amount needed to widen the columns that will allow the row height to be decreased by one line. We show the required column widening for each row by the grey box in the appropriate cell in the row. The algorithm will choose to widen the second column since this reduces the height of the second row (and hence table) for the smallest increase in table width.



(d) The layout after widening the second column. The algorithm recomputes the amount needed to widen the columns that will allow the row height to be decreased by one line. This is shown by the grey boxes. Again it will choose to widen the second column since this reduces the height of the second row (and hence table) for the smallest increase in table width.



(e) The layout after widening the second column again. The grey box shows the narrowest width for each cell that allows the row height to be decreased by one line. The main loop of the Iterative Column Widening Algorithm terminates since it is not possible to reduce the height of any row by widening the table except by widening it beyond its maximum width. Thus this is the final layout.

Figure 4: Example of how iterative column widening can be used to improve the table layout generated by the Area Approximation Algorithm. (Text is from the start of Aesop's fable "The Ugly Duckling.")

`walk`^{6} `about`^{7} `in`^{8}
`the`^{9} `country.`^{10}

We have that $cwidth = 15$, $lastLine = 4$ and $lineStart = [1,4,6,9]$. The base case will compute *minWidth* for the words in the last two lines: in this case the only way to reduce the number of lines is to place the remaining words in a single line. Thus, *minWidth* is set to the length of this line if it is greater than *cwidth*, otherwise to *cwidth*. We have that $minWidth[10] = minWidth[9] = minWidth[8] = 15$, $minWidth[7] = 21$ and $minWidth[6] = 26$.

The algorithm now computes *minWidth* for the words in the second line. w is set to the start of line 2, i.e. 4, and nlw to the start of line 3, i.e. 6. Basically the idea is to walk nlw along line 3 to find the best possible line break to end the line starting at w. *lineLength* is initialised to the length of the line between w and nlw i.e. the length of line 2 ignoring trailing spaces. This is 13. Now $minWidth[nlw] = 26$ so that this tells us that if we placed the line break at $nlw = 6$ the width required would be $max(13,26) = 26$. We now move nlw along the line to word 7 and appropriately increase *lineLength* by $1 + width[6] = 5$ to give $lineLength = 18$. Since $minWidth[7] = 21$ this tells us that if we placed the line break at $nlw = 7$ the width required would be $max(18,21) = 21$. We now try $nlw = 8$. We increment *lineLength* by $1 + width[7] = 6$ to give $lineLength = 24$. Since $minWidth[8] = 15$ this tells us that if we placed the line break at $nlw = 8$ the width required would be $max(24,15) = 24$. Since $lineLength \geq minWidth[nlw]$ we can stop our search since all subsequent values will be worse since they will have larger values for *lineLength*. Thus, $minWidth[w] = minWidth[4]$ is set to $min(21,24) = 21$. We now increment w to 5 and appropriately reduce *lineLength* by $1 + width[4]$ so that *lineLength* continues to be the length of a line from word w to word nlw. Thus *lineLength* is $24 - 11 = 13$. We do not need to reset nlw to the start of line 3 since we can show that the best line break for a line starting at $w = 5$ will always be

next-shorter-cell-configuration()
/* base case merge last line into previous line */
$lineLength := width[n]$
for $w := n$ **downto** $lineStart[lastLine-1]$ **do**
  $minWidth[w] := max(cwidth, lineLength)$
  $lineLength = lineLength + 1 + width[w-1]$
end for
for $L := lastLine - 2$ **downto** 1 **do**
  $nlw := lineStart[L+1]$ /* pointer along line $L+1$ */
  /* Initialize $lineLength$ to the width of line L. */
  $lineLength := width[lineStart[L]]$
  for $w := lineStart[L]+1$ **to** $lineStart[L+1]-1$ **do**
    $lineLength := width[w] + 1 + lineLength$
  end for
  for $w := lineStart[L]$ **to** $lineStart[L+1]-1$ **do**
    /* $lineLength$ is the width of words $w, \ldots, nlw-1$. */
    while $minWidth[nlw] > lineLength$ **and**
        $nlw < lineStart[L+2]$ **do**
      $lineLength := lineLength + 1 + width[nlw]$
      $nlw := nlw + 1$
    end while
    /* Choose whether next line starts at word $nlw-1$ or nlw. */
    $minWidth[w] := min(minWidth[nlw-1], lineLength)$
    $lineLength := lineLength - (1 + width[w])$
  end for
end for
return $minWidth[1]$

Figure 5: Algorithm to compute next shorter minimal configuration for a cell

greater than or equal to that for a line starting at $w = 4$. For the current value of $nlw = 8$ the width required is $max(13,15) = 15$. Since $13 < 15$ we increment nlw to 9 and increase $lineLength$ to 16. Since $nlw = lineStart[4]$ the while loop terminates and the $minWidth[w] = minWidth[5] = min(15,16) = 15$.

The algorithm now processes the words in the first line in a similar fashion, obtaining $minWidth[1] = 21$, $minWidth[2] = 21$ and $minWidth[3] = 18$. Thus the next shorter minimal configuration has width 21. It is

```
It was, indeed,
delightful to walk
about in the country.
```

7. EVALUATION

In this section we compare our four layout algorithms with each other and also with HTML table layout provided in the Mozilla browser. We used the proprietary Mosek library [2] for solving the continuous approximation of the problem and a memoization-based variant of the algorithm in Figure 5 to compute the minimum cell width for a particular height. All experiments were conducted on a 1.2GHz AMD Athlon with 1.5GB RAM.

In our first experiment we took the seven tables shown in Figures 1, 6, 7 and 8 and laid out the tables using our layout algorithms and Mozilla. All were laid out with a table width of 800 pixels and automatic sizing of columns.

For each of our four methods we measured the time required to layout our example tables. In the case of the hybrid approaches we give two times: that for computing the initial column widths using area approximation and our variant of the HTML algorithm, respectively, and the time taken in the subsequent layout improvement using iterative column widening. The layout times exclude time taken for table parsing and final rendering. All times are in milliseconds. The results are shown in Table 1. We have not given times for Mozilla's algorithm; we would expect similar times to those given for our variant of the HTML algorithm, i.e. considerably faster than any of the more complex algorithms presented in this paper.

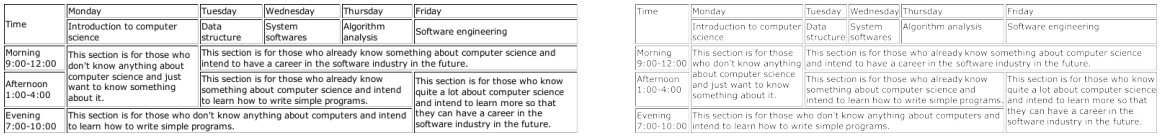
Example	AA	ICW	AA+ICW		HTML-V+ICW	
			AA	ICW	HTML-V	ICW
2n2-linear	106	3	106	0	0	0
multipara	106	13	107	0	0	0
simple-brick	108	3	107	0	0	0
cs-schedule	111	3	111	0	0	0
diagonal5	109	3	110	0	0	1
columns	116	322	108	12	1	19
plants200	2199	7183	2178	1192	3	424

Table 1: Time taken by Area Approximation (AA), Iterative Column Widening (ICW), the first hybrid approach (AA+ICW) and the second hybrid approach (HTML-V+ICW) to layout the example tables. All times are in milliseconds.

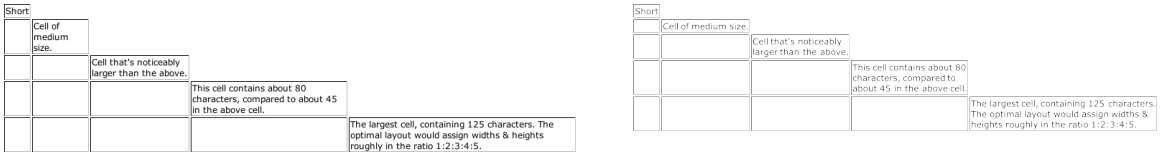
Figure 7: The example *columns* has uneven height multi-row spanning columns and is a case where we would expect the simple heuristic used in iterative column widening to perform badly. Layout using Mozilla is on the left that with the first hybrid approach on the right.

We see that iterative column widening is considerably faster than the area approximation approach in all but the last two examples. This is not too surprising, since conic programming has substantial overhead and so can be expected to be relatively slow when solving small problems while the second last example, *columns*, was chosen as a pathological case for iterative column widening and in the case of the last example, *plants200*, the large number of rows slows iterative column widening considerably.

As one would expect starting iterative column widening from initial column widths computed using either area approximation or our HTML-variant significantly reduces the time taken for iterative column widening. Of course our second hybrid version which uses the HTML-variant is faster than the first using area approximation. Indeed, the hybrid method using our HTML variant is the fastest method and handles even large tables like *plants200* in less than 0.5 seconds.



Example *cs-schedule* is based on an example from [11], showing a cycle among compound cells



Example *diagonal5* has a simple solution to its continuous approximation, and was chosen to test how closely this continuous solution matches the final solution

Figure 6: Two more example tables comparing layout using Mozilla (on the left) with our first hybrid approach (on the right).

Example	Mozilla	HTML-V	HTML-V+ICW	AA	ICW	AA+ICW
2n2-linear	63	63	63	63	63	63
multipara	209	209	150	165	150	150
simple-brick	128	70	70	70	70	70
cs-schedule	177	177	177	177	177	177
diagonal5	205	205	176	191	176	176
columns	1135	1134	1076	1031	1017	1017
plants200	6150	6150	5776	6200	5550	6086
2n2-linear	100.0%	100.0%	97.5%	100.0%	97.5%	97.5%
multipara	100.0%	100.2%	72.9%	75.1%	72.9%	72.9%
simple-brick	100.0%	66.4%	62.7%	63.6%	62.7%	62.7%
cs-schedule	100.0%	102.5%	91.8%	97.2%	91.8%	91.8%
diagonal5	100.0%	100.0%	87.6%	99.6%	87.9%	87.6%
columns	100.0%	100.5%	93.5%	90.7%	89.3%	89.6%
plants200	100.0%	100.2%	96.8%	99.0%	94.6%	97.4%

Table 2: Height (in pixels) of tables when using Mozilla, our HTML variant (HTML-V), Area Approximation (AA), Iterative Column Widening (ICW) and the two hybrid approaches to layout the example tables; the first block shows heights with a requested width of 800px; the second block shows the average over widths 400px, 450px,. . . ,1200px of the proportion of Mozilla's height for the same width.

Next we compared the quality of layout of our four methods with each other, with Mozilla and with our variant of the HTML algorithm. In Table 2 we give the table height in pixels for each of the methods and in Figures 1, 6, 7 and 8 we show the layout obtained with Mozilla and the first hybrid approach.

One possible criticism of this experiment is that the outcomes may be too dependant on the exact table width. For this reason we conducted a third experiment in which we compared the quality of layout for the different methods on 17 different table widths. The results are summarized in the second part of Table 2.

We find that our four methods give comparable layout and for many of the examples it is significantly more compact than that found by Mozilla. Our HTML variant gives better layout than Mozilla for multi-column cells and similar layout otherwise. It is not as compact as that produced by the four more complex methods. Area approximation gives the worst layout out of the other methods, giving less compact layout on *diagonal5* than the other methods (but one could argue that the table looks more symmetric). Arguably iterative column widening gives the best overall layout. However, it is worth observing that if hyphenation is allowed then we would expect the area approximation to be more accurate.

8. CONCLUSION

Despite the importance of table layout in document formatting, there has been relatively little work on automatic formatting of tables. We have described two new techniques for automatic layout, area approximation and iterative table widening and two hybrid techniques. We believe these techniques will have application in web-browsers, document formatting systems and even spreadsheets.

All of our techniques give better and arguably more robust table layout than current HTML table layout engines with iterative column widening giving the best layout in the sense that the table height is minimised for a particular width. However for large tables iterative column widening can be slow. For this reason probably the hybrid approach combining iterative column widening with a variant of the HTML algorithm is the most practical of our approaches, taking less than 0.5 seconds even for very large tables, yet producing quite compact layout.

Our approach to solving the continuous table layout problem is relatively slow for small tables because of the overhead of conic programming. However, we believe that it is possible to solve the continuous table approximation problem more quickly than Mosek does. We are currently investigating a specialised active set based method for solving the continuous table layout problem and also

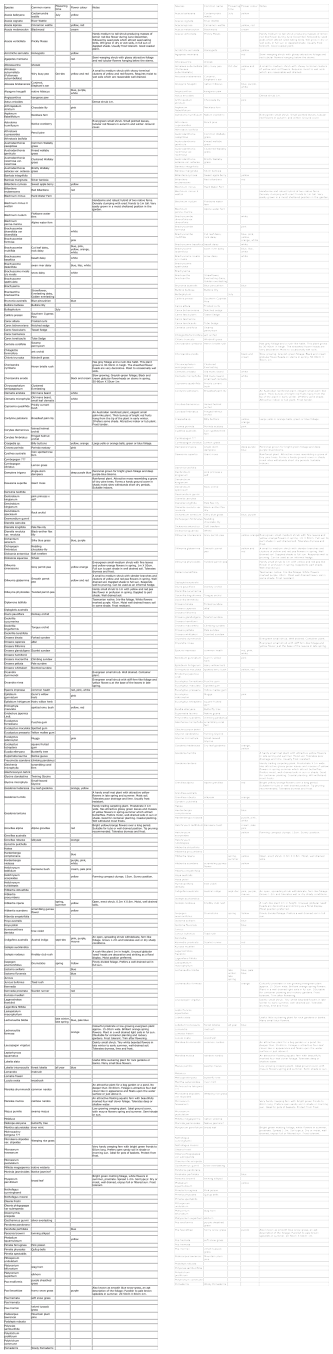


Figure 8: Last example comparing layout using Mozilla (on the left) with our first hybrid approach (on the right). This example, *plants200*, is a real-world example of a plant database displayed as a large table with 200 rows.

heuristics for finding a "good" solution quickly. We believe the ideal of approximating text by its area has other applications, such as approximate page layout. We are also investigating this.

9. REFERENCES

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

[2] M. ApS. Mosek optimization toolkit. Web page, 2005. `http://www.mosek.com/products_mosek.html`.

[3] 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.

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

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

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

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

[8] D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. In *Software—Practice and Experience, 11(11)*, pages 1119–1184, Nov. 1982.

[9] 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.

[10] J. Renegar. *A Mathematical View of Interior-Point Method in Convex Optimization*. SIAM, 2001. [Edital Universal, 2001].

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