Automatic Float Placement in Multi-Column Documents

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

ABSTRACT

Multi-column layout with horizontal scrolling has a number of advantages over the standard model (single column with vertical scrolling) for on-line document layout. However, one difficulty with the multi-column model is the need for good automatic placement of floating figures. We identify reasonable aesthetic criteria for their placement, and then give a dynamic-programming-like algorithm for finding an optimal layout with respect to these criteria. We also investigate an $A^\star$ based approach and give two variants differing in the choice of heuristic. We find that one of the $A^\star$ based approaches is faster than the dynamic programming approach and, if a "window" of optimization is used, fast enough for moderately sized documents.

Categories and Subject Descriptors

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

General Terms

Algorithms

Keywords

optimization techniques, floating figure, multi-column layout

1. INTRODUCTION

The standard layout model for on-line textual documents is to lay text out in a single column whose width is that of the viewing window. If the text does not fit in the window, vertical scrolling is used to read the rest of the document. An alternate model for on-line layout is a multi-column layout with horizontal scrolling. In this model, the text is placed in columns whose height is that of the viewing window and whose width is fixed but perhaps based on font-size or on user preferences. Columns are added as needed to the right[1] of the current column. If the text does not fit in the window then horizontal scrolling is used to read the rest of the document, rather than vertical scrolling. See, for example, Figure 1.

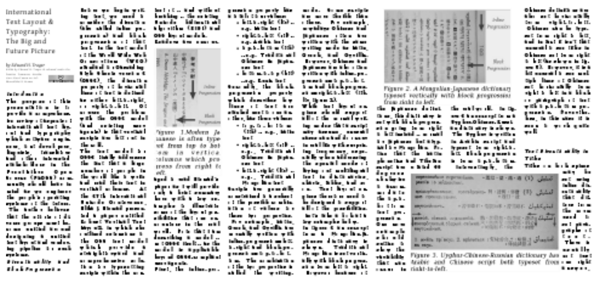
Figure 1: Example of a multicolumn document as formatted by our software.

Such horizontal scrolling multi-column layout has two main advantages over single column layout. First, the column width can be chosen to make the document more pleasant to read, rather than having the text width set to some percentage of the browser width.[2] Second, multiple columns allow for more interesting and aesthetically pleasing layout—very few magazines use a single column layout. Two, more minor, advantages are that multi-column layout allows footnotes to be placed at the bottom of the column rather than at the end of the document and navigation information such as section and chapter headings to be displayed at the top/bottom of the window.

On the surface, implementing multi-column layout does not seem that difficult: However, as we now discuss, placement of floating figures (which we shall refer to as floats) is considerably more difficult than in the single column model. In the (X)HTML or CSS single column layout mode, the author specifies a single "placement style" for each float (inline, left or right) and the float is rendered with that style when the float is first encountered while rendering the doc-

[1]The words "left" and "right" throughout this paper assume that columns are ordered left-to-right and text flows downward as in English; for right-to-left columns, swap their meanings.
[2]One rule of thumb, at least for print, is that the column width (or measure) should be no less than 27 characters, ideally 40 characters, and no more than 70 characters [2]

ument. As long as the text line width is similar to that intended by the author this works reasonably well.

Unfortunately, this simple model does not generalize well to multi-column layout. One of the advantages of multiple columns is that they allow more varied and aesthetically pleasing placement of floats. Floats can span more than one column and may "eat" into columns. This richer choice of placement styles makes it more difficult for the author to choose a single best style. Furthermore, in the multi-column model the choice of float placement style is more brittle and so it is not uncommon to require different choices for different viewing environments. Thus, it is not reasonable to require the author to specify a single best layout style for each float and a single best point in the document at which to render the float.

We believe a better approach is for the layout engine to provide more powerful automatic float placement. In the model detailed here the author can indicate a list of allowed placement styles for each float, but the layout engine is responsible for the choice of style and placement of the float. Each float can be associated with text that it is logically connected to and the layout will attempt to place floats as close as possible to their associated text. The layout engine also takes into account reasonable aesthetic criteria such as a minimum sufficient gap between floats and that the "reading order" of the floats should reflect their underlying order. We believe that by forcing the author to give the logical structure of the document and its floats rather than specifying their precise layout, this allows better layout that truly adapts to different viewing contexts.

This is one of the first papers to investigate automatic float placement in multi-column documents. We believe our work has relevance to document processing and web standards. One of the widely recognized limitations of the recent web document formatting standards (X)HTML and CSS is lack of support for multi-column text layout, leading to proposals to extend (X)HTML and CSS to support it [6]. Similarly there are proposals to provide better support for automatic float placement in TEX, for instance [7]. We hope that the current paper will provide some guidance for these efforts.

The main contributions of this paper are: to identify the need for automatic float placement when laying out multi-column on-line documents; to suggest a number of reasonable aesthetic criteria for automatic float placement (Section 2); to give algorithms for automatic placement (Sections 3 and 4); and to provide an empirical evaluation of these algorithms (Section 5).

We first give a dynamic programming based algorithm to find an optimal layout. This starts from the empty layout and repeatedly extends this by adding either a line of text or a figure. The algorithm works by constructing larger and larger partial layouts, at each step choosing to expand the smallest layouts. Thus, it performs a breadth-first exploration of the search space consisting of the partial layouts.

Another approach to exploring the search space is a best-first traversal in which the most promising partial layout is repeatedly chosen and expanded. The potential advantage of a best-first traversal is that with a good choice of ranking heuristic the algorithm can avoid expanding bad partial layouts that will not lead to an optimal solution. One of the most widely used algorithms for complete best-first search for combinatorial problems is the A* algorithm [9]. We present two $A^\star$ based algorithms that vary in the choice of heuristic used to rank partial layouts. Both are guaranteed to find an optimal solution. Our empirical evaluation shows that one of the $A^\star$ based algorithms is significantly faster than the dynamic programming algorithm and fast enough for small documents.

Finally, we give a modification to these $A^\star$ based algorithms that means the algorithms are much faster but not guaranteed to find the optimal solution. The idea is that a "window" of optimization is used meaning that after certain point earlier figure placements are not reconsidered. In practice these modified algorithms still find very good solutions and so are suitable for larger documents.

There has been relatively little research into placement of floats. The only problem that has been studied in detail is float placement in paged documents where the floats are placed at the top (and/or bottom) of the page and are effectively as wide as the page. The objective of the layout is to ensure that each float occur on the same page as the first reference to it or on a following page as close as possible to the reference. The main decisions in the layout are on which page to place each float and how much white space to leave at the bottom of each page.

Plass [8] investigated how the precise formulation of this problem affected its difficulty. He showed that if a quadratic penalty function is used then the problem is NP-hard, but that if a linear penalty function is used then the optimal layout can be computed in polynomial time using a dynamic programming algorithm. Brüggemann-Klein et al. [1] further investigated the use of dynamic programming for this problem and we believe that the system of Jacobs et al. [5] extended the dynamic programming algorithm to handle templated pages in which figures could occur at a fixed locations on the page, not just the top/bottom. In practice, however, the dynamic programming algorithm is quite slow and even slower if the line length depends upon figure placement such as in the case text is allowed to wrap around figures.

The type setting system LATEX uses a greedy heuristic for float placement. It does not handle floats and multi-column layout or floats "wrapped" by text very well at all. An extension to handle float placement in multi-column layout has been proposed by Mittelbach [7]. This also utilizes a greedy heuristic.

2. FLOAT PLACEMENT MODEL

Our underlying layout model is that the canvas is composed of horizontally adjacent columns of fixed width and height with a fixed-width gap between the columns. The text is a sequence of words $w_1, ..., w_m$ and there are associated floats $F_1, ..., F_n$. Floats are allowed to be wider than a single column. Text flows into the space not occupied by floats.

We must first understand the general aesthetic criteria for judging float placement. We have identified seven aspects:

AC1 Non-overlap and containment: Floats should be contained in the canvas and must not overlap each other.

AC2 Placement style: This specifies the allowable positions for each float on the canvas. Typical styles are left- or right-aligned on a column, centered in a column, or centered on a column gap.

AC3 Floats should be placed near their associated text.

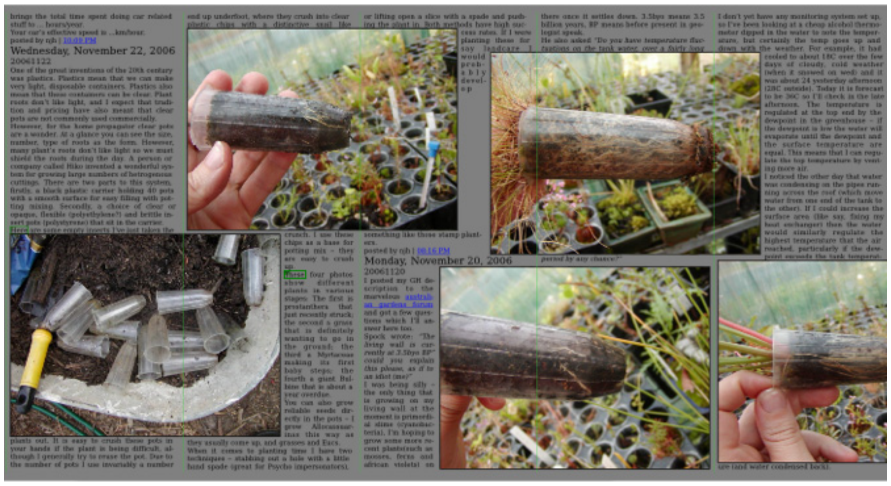

Figure 2: Example of bad placement of floats in multi-column layout. This example illustrates how bad float placement can separate text in the columns so that it becomes difficult to find the next line when reading (AC5)–consider for instance finding the last line of text in the fourth column. The floats are also quite close together and look cluttered (AC6) and the layout contains unnecessary white space in column three (AC7).

AC4 The "reading order" for the floats should correspond to the order in which they are referred to in the text so as to allow the reader to more readily find the float. By the reading order we mean the order in which the floats are first encountered when reading the text.

AC5 Float placement does not lead to bad breaks in the text. One aspect is bad line breaking, for instance breaking a heading in two, and the other is separating text so that it is difficult to find the next line of text. See for example Figure 1.

AC6 Floats are distributed "nicely" throughout the canvas. There needs to be adequate space between figures so that they are not cluttered.

AC7 The layout is compact and tries to avoid unnecessary white space.

The preceding aesthetic criteria are quite vague and imprecise. In order to develop automatic float placement algorithms, we need to determine which criteria are to be rigidly enforced and give an exact specification of the penalties and trade offs between those remaining.

In our model, for simplicity, we assume that each float F_j is a fixed-size rectangle with height H_j and width W_j (including any associated caption). Non-overlap and containment of each float within the page (AC1) is enforced in the model (unless the float is too high for the column, in which case it is aligned to the top of a column and extends below the bottom of the desired column height). We allow floats to extend past the end column, but not to extend before the first column.

The placement style for each float F_j is modelled by a non-empty list of allowed horizontal justifications—left or right aligned on a column, centered in a column, or centered on a column gap. Note that for each of these the float can span more than one column. Only the placement styles specified in the lists are considered for the float and, if the quality of the layout is the same, the placement style occurring earlier in the list is preferred. A column is allowed to be eaten into on the left and on the right at the same level. However, a float centered in a column is treated as if it has the column width.

There is a unique preferred position R_j for the float in the text sequence. This means that we want to place F_j close to the word w_{R_j}. Typically, R_j is the index of the word first referring to the float. We assume that the floats are ordered by their reference, i.e. $R_{j+1} > R_j$ for all $j = 1, ..., n-1$. It is unclear (at least to the authors) how best to measure the distance between the position of a float F_j and the position of the word w_{R_j} when computing the penalty for aesthetic criterion AC4. In this paper, we use the (absolute) distance in column inches between the top of F_j and the top of the text line containing w_{R_j}. Other reasonable possibilities would be to measure from the middle or bottom of the figure and/or to use the Euclidean distance rather than column inches.[3]

In our model, floats are placed in consecutive order. This ensures that the reading order exactly mirrors the order in which they are referred to (AC4). This reduces the complexity of automatic float placement but limits some flexibility in the layout process: e.g. if a large float needs to go to a future column, then a human might consider placing a subsequent smaller float before the large float so that the small float can be in the same column as the text that refers to it.

For simplicity our model does not directly address aesthetic criteria AC5 or AC6. In Section 6 we discuss how we have extended it to handle these.

Criterion AC7 is (partially) addressed by penalizing extra white space at the end of text lines using a penalty function similar in spirit to that of TeX. One difference is that we discretize the penalty; in particular, low penalties are rounded down to 0 and high penalties rounded to 1.

[3]However these alternatives require adding more context to the "frontier" (F) used in the algorithms described in Sections 3 and 4, and so presumably will lead to additional computational cost when using these approaches.

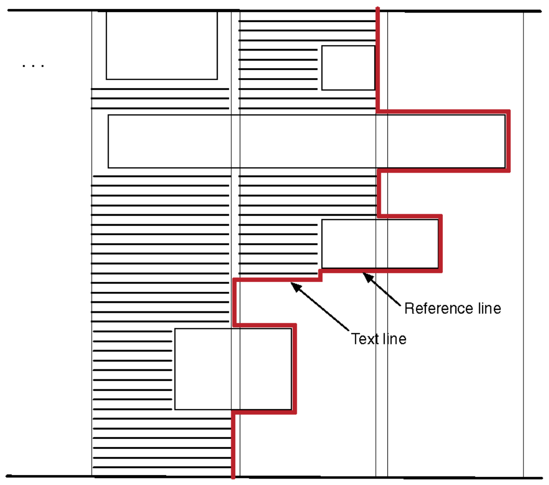

Figure 3: Frontier for an example partial layout.

An interesting question is how the different penalty terms are combined into a single global penalty. Currently we use a weighted linear combination but plan to investigate other combinations.

3. DYNAMIC PROGRAMMING APPROACH

We now investigate algorithms for determining an optimal layout for a document. That is, one that has a minimal penalty while satisfying the above layout constraints. Note that more than one optimal layout may exist. Since Plass [8] and Brüggemann-Klein et al. [1] have previously used dynamic programming to solve a related float placement problem, we first consider a dynamic programming-like algorithm which is similar in spirit to their algorithms, although considerably more complex. The main reason for the increased complexity in the algorithm is the need to handle figures spanning more than a single column.

The algorithm works by constructing *(partial) layouts* and extending these by either adding a line of text or a figure to give a new partial layout. A partial layout is represented by a state $\langle i, j, F, p, h\rangle$ that consists of the index i of the last word in the layout, the index j of the last figure, the frontier F of the layout, the penalty p for the layout, and h a reference to the partial layout this was obtained from.

The frontier contains all the state necessary (other than (i, j)) to calculate the set of valid extensions (future partial layouts, defined more precisely below), what their penalty relative to p is, and, in turn, what their frontier is. For the aesthetic criteria and penalty function that we use in this paper, the frontier is a specification of the region available for future text and floats, and may be specified by its boundary line, a sequence of alternating downward and horizontal line segment lengths starting from the top-right of the current column; an example is shown in Figure 3.

There is a single distinguished horizontal line in the frontier called the *text line*, which is where the top of the next text would go (if there were more text, and if that text were small enough to fit at that place, and if we were to place that text instead of placing a figure). The text line is unique, because we do not allow text in a single column to flow around both sides of a figure—it either goes on the left or on the right or neither. Two frontiers are regarded as different if they have different text lines even if they are otherwise identical. We are also interested in the *reference line* of the frontier, which is the highest horizontal line in the frontier crossing the current column. Clearly, this must be above the first point that any figure or text could be placed in a subsequent layout.

If P is a *complete* layout containing all of the text and figures, i.e. $i = m$ and $j = n$, then its penalty is given by

$$p(P) = ws_P + \sum_{k=1}^{n} |F_k^P - w_{R_k}^P|$$

The first component, ws_P, is a non-negative penalty for whitespace, bad line breaking etc. in the layout so far; while the second is the placement penalty for the figures, where F_k^P and $w_{R_k}^P$ are, respectively, the distance in column inches from the top of F_k and the top of the line containing w_{R_k} to the reference line of P.[4]

We also need to give a penalty for incomplete layouts. A partial layout $P = \langle i, j, F, p, h\rangle$ implicitly partitions the figures into 4 sets: those which are fixed and their reference is fixed,

$$FF_P = \{k \mid k \leq i \wedge R_k \leq j\};$$

those which are fixed but their reference is not,

$$FU_P = \{k \mid k \leq i \wedge R_k > j\};$$

those which are not fixed but their reference is,

$$UF_P = \{k \mid i < k \wedge R_k \leq j\};$$

and those which are not fixed and nor is their reference,

$$UU_P = \{k \mid i < k \wedge R_k > j\}.$$

For a partial layout P, we use penalty function $s(P)$, defined as:

$$ws_P + \sum_{k \in FF_P} |F_k^P - w_{R_k}^P| + \sum_{k \in FU_P} F_k^P + \sum_{k \in UF_P} w_{R_k}^P.$$

The first component, ws_P, is a non-negative penalty for whitespace, bad line breaking etc. in the layout so far. The remainder is the placement penalty for figures. If figure F_k and w_{R_k} have both been placed in the layout, then the penalty component for that figure is $|F_k^P - w_{R_k}^P|$; if neither F_k nor w_{R_k} have been placed in the layout, then that component is 0; if F_k has been placed in the layout but not w_{R_k}, then this component is F_k^P the distance from the top of F_k to the reference line of P. The remaining case is dual.

Note that if P is a complete layout, then all figures and references are placed, so

$$s(P) = p(P) = ws_P + \sum_{k=1}^{n} |F_k^P - w_{R_k}^P|. \tag{1}$$

The unprocessed partial layouts are kept in $m \times n$ priority queues, $PQ_{i,j}$. Each priority queue $PQ_{i,j}$ contains those partial layouts of form $\langle i, j, F, p, h\rangle$ and is ordered by

[4]It does not matter where distance is measured from since it is only the difference between distances that are included in the penalty. We choose to measure it from the reference line of the layout as this is more convenient when extending the penalty to partial layouts.

p. The algorithm starts by placing the *empty* partial layout $\langle 0,0,F,p,h\rangle$ in the priority queue $PQ_{0,0}$, where $p = 0$, $h =$ NULL and $F = [-col\_width, col\_height]$ where *col_width* and *col_height* are the column width and height respectively. All other priority queues initially have no elements. The algorithm processes each of the priority queues, $PQ_{i,j}$, in order of increasing $i + j$.

If $i = m$ and $j = n$ the priority queue contains only complete layouts. In this case the complete layout P with minimum penalty is removed from the priority queue and the algorithm terminates since P is an optimal layout. The reason for including h in the layout record is that it allows the algorithm to determine the placement of text and figures that lead to this optimal layout, since the algorithm can follow the references backwards to find the sequence of intermediate partial layouts used to construct this layout and so reconstruct the entire layout.

Otherwise, if $i < m$ or $j < n$, the algorithms works by repeatedly removing the partial layout $P = \langle i,j,F,p,h\rangle$ with the smallest value of p from the priority queue $PQ_{i,j}$. We first check that we have not already processed a layout from $PQ_{i,j}$ with the same frontier. This is done by keeping a *closed set* (sometimes called a *closed list*) of frontiers for each priority queue. If F is already present in the closed set, then we have already expanded an equivalent partial layout whose penalty is the same or lower, so we discard P and go on to the next item in the queue.

The algorithm constructs all partial layouts $P_1, ..., P_k$ that are *valid extensions* of P. These are obtained by either adding a line of text starting with word w_{i+1} or placing the figure F_{j+1}. For the partial layout to be a valid extension, the top of the newly-placed figure or line of text must be below or equal to the top of the most recent previously-placed figure or line of text (if any); also (if placing a figure), the figure F_{j+1} must be in an allowed placement, and either there is no more text to place or adding the line of text precludes placing the figure F_{j+1} at that position. The partial layouts $P_1, ..., P_k$ are added to the appropriate priority queue and the next iteration begins. Note that because each $P_1, ..., P_k$ either increases the number of words or figures in a layout they will be placed in priority queues that has not yet been processed.

An *allowed layout* is a complete layout that can be obtained from the empty layout by a sequence of valid extensions. An allowed layout is *optimal* if there is no other allowed layout P with smaller penalty $p(P)$.

Correctness of the dynamic programming algorithm relies on the observation that:

Lemma 3.1. *Let $P = \langle i,j,F,p,h\rangle$ and $P' = \langle i,j,F,p',h'\rangle$ be partial layouts. Then (a) If $p < p'$, then P' cannot be extended to give an optimal layout, and (b) If $p = p'$, then P' can be extended to give an optimal layout iff P can be.*

Proof. We prove (a). The proof for (b) is similar. The proof is by contradiction. Assume that P' can be extended to give an optimal layout, Q', for the document. Since P and P' have the same frontier and text line, the same operations used to extend P' are also valid extensions for P. Let Q be the layout resulting if these extensions are applied to P. By construction Q is a complete valid layout.

We now prove that the penalty, q, for Q is less than the penalty, q', for Q'. Now

$$\begin{aligned} q &= ws_Q + \sum_{k=1}^{n} |F_k^Q - w_{R_k}^Q| \\ &= ws_Q + \sum_{k \,\in\, (FF_P \,\cup\, FU_P \,\cup\, UF_P \,\cup\, UU_P)} |F_k^Q - w_{R_k}^Q| \end{aligned}$$

Since Q extends P, the placement for figure F_k is the same in both Q and P for $k \leq j$ and also for words w_k for $k \leq i$ modulo the change in reference line. Thus, p is

$$ws_P + \sum_{k\in FF_P} |F_k^Q - w_{R_k}^Q| + \sum_{k\in FU_P} \left(F_k^Q - b^{PQ}\right) + \sum_{k\in UF_P} \left(w_{R_k}^Q - b^{PQ}\right)$$

where b^{PQ} is the distance from the reference line of P to that of Q. Since the top of figures and text added to extend P must occur after P's reference line we have that for all $k \in FU_P$,

$$|F_k^Q - w_{R_k}^Q| = F_k^Q - w_{R_k}^Q = (F_k^Q - b^{PQ}) + (b^{PQ} - w_{R_k}^Q)$$

Similarly, for all $k \in UF_P$,

$$|F_k^Q - w_{R_k}^Q| = w_{R_k}^Q - F_k^Q = (w_{R_k}^Q - b^{PQ}) + (b^{PQ} - F_k^Q).$$

Thus, q is

$$p + ws_r + \sum_{k\in FU_P} (b^{PQ} - w_{R_k}^Q) + \sum_{k\in UF_P} (b^{PQ} - F_k^Q) + \sum_{k\in UU_P} |F_k^Q - w_{R_k}^Q|$$

where ws_r is the penalty for white space, line breaking etc. for the components not laid out in P.

Similarly, q' is

$$p' + ws_{r'} + \sum_{k\in FU_{P'}} (b^{P'Q'} - w_{R_k}^{Q'}) + \sum_{k\in UF_{P'}} (b^{P'Q'} - F_k^{Q'}) + \sum_{k\in UU_{P'}} |F_k^{Q'} - w_{R_k}^{Q'}|$$

where the terms are defined analogously.

Now P and P' fix exactly the same words and figures so $FF_P = FF_{P'}$ etc. And since Q and Q' have been extended using the same operations from the same frontier $b^{P'Q'} = b^{PQ}$ and so

$$\begin{aligned} ws_r &= ws_{r'}, \\ \sum_{k\in FU_P} (b^{PQ} - w_{R_k}^Q) &= \sum_{k\in FU_{P'}} (b^{P'Q'} - w_{R_k}^{Q'}), \\ \sum_{k\in UF_P} (b^{PQ} - F_k^Q) &= \sum_{k\in UF_{P'}} (b^{P'Q'} - F_k^{Q'}), \\ \sum_{k\in UU_P} |F_k^Q - w_{R_k}^Q| &= \sum_{k\in UU_{P'}} |F_k^{Q'} - w_{R_k}^{Q'}|. \end{aligned}$$

By hypothesis $p < p'$ and so $q < q'$. Thus, Q is an allowed layout with lower penalty than Q', contradicting the initial assumption that Q' is an optimal layout. □

Theorem 3.2. *The dynamic programming algorithm will terminate and return an optimal allowed layout.*

Proof. Clearly the algorithm terminates since: (a) there are a finite number of priority queues and it processes each priority queue $PQ_{i,j}$ once by removing and processing all partial layouts in the priority queue, and (b) processing of a partial layout in $PQ_{i,j}$ only adds elements to unprocessed priority queues and can never add an element to $PQ_{i,j}$.

The algorithm returns an allowed layout because it only returns complete layouts, and all the layouts it generates are obtained from the empty layout by a sequence of valid extensions.

When it terminates it returns the complete layout P with the minimum penalty $s(P)$. This is because the algorithm generates all allowed layouts except that it discards partial layouts that cannot lead to a minimal penalty or for which there is already a layout which will lead to the same penalty (by Lemma 3.1). From Eqn. 1, $s(P) = p(P)$ for complete layouts, so P is an optimal allowed layout. $\square$

What is less clear, is the complexity of the dynamic programming algorithm. Computing the valid extensions to a partial layout and placing and removing objects from the priority queues have polynomial complexity. Thus, the algorithm will have polynomial complexity if we can prove that only a polynomial number of partial layouts are expanded. Clearly a sufficient condition for this is that the number of possible $\langle i, j, F\rangle$-tuples is polynomial. Unfortunately, in general because of the frontier this is not true. However, for various restrictions it is true. For instance, if figures span at most a single column and can only be placed in the center of a column and at the top or bottom of the page, then there are only a polynomial number of frontiers. Or, if line heights are all equal, figure heights are a multiple of the line height, and only a bounded number of figures can extend to the right of the current column, then the number of frontiers is polynomial.

4. AN A*-BASED APPROACH

The dynamic programming algorithm works by constructing larger and larger partial layouts. Before constructing a layout with i words and j floats, it must first exhaustively consider all partial layouts with fewer words and fewer floats. Thus, the algorithm performs a breadth-first exploration of a search space consisting of the partial layouts where a partial layout can be reached from another partial layout if it is a valid extension. A different approach is a best-first traversal in which the most promising partial layout is repeatedly chosen and expanded. The advantage is that the algorithm can avoid expanding bad partial layouts that will not lead to an optimal solution.

One of the most widely used algorithms for complete best-first search for combinatorial problems is the A* algorithm [9]. In our context this works by associating a priority $f(P) = g(P) + h(P)$ with each partial layout P where $g(P)$ is the penalty accrued so far and $h(P)$ is a conservative heuristic estimate of the penalty for extending P to a complete layout.

Rather than using $m \times n$ priority queues, the A* algorithm uses a single priority queue ordered by the penalty. The algorithm starts by placing the empty partial layout in the priority queue. The algorithm works by repeatedly removing a partial layout $P = \langle i, j, F, p, h\rangle$ with the smallest value of p from the priority queue where p is now $f(P)$. If P is complete, then P is an optimal layout and so the algorithm stops. Otherwise, we first try to insert $\langle i, j, F\rangle$ into the *closed set* (the set of previously expanded partial layouts). If it is already present, then we have already expanded a layout with the same set of possible extension sequences and corresponding penalty additions and with the same or lower existing penalty; so we can discard P and go on to the next item in the queue. Otherwise, all valid partial extensions of P are constructed and added to the priority queue.

Correctness of the A* algorithm relies on the penalty $f(P)$: (a) being equal to the actual penalty $p(P)$ for complete layouts, and (b) being *monotonic* (or *consistent*), in the sense that if partial layout Q is a valid extension of partial layout P, $f(P) \le f(Q)$. The penalty function $s(P)$ given earlier satisfies these requirements. Requirement (a) follows from Eqn. 1.

LEMMA 4.1. *The penalty function $s(P)$ is monotonic.*

PROOF. Let $s_k(P')$ be the penalty associated with figure k and its reference in partial layout P' and ws'_P the penalty for whitespace.

We must prove that if partial layout Q is a valid extension of partial layout P, $s(P) \le s(Q)$. Now $s(P) = ws_P + \sum_{k=1}^{n} s_k(P)$ and $s(Q) = ws_Q + \sum_{k=1}^{n} s_k(Q)$. Since Q extends P, $ws_P \le ws_Q$.

We now prove that for all k, $s_k(P) \le s_k(Q)$. There are 8 cases to consider:

- If $k \in UU_Q$ then $k \in UU_P$ and so $s_k(P) = s_k(Q) = 0$.
- If $k \in FF_P$ then $k \in FF_Q$ and so $s_k(P) = s_k(Q)$.
- If $k \in UU_P \cap FU_Q$ then $s_k(P) = 0$ and $s_k(Q) = F_k^Q \ge 0$.
- If $k \in FU_P \cap FU_Q$ then
$$s_k(Q) = F_k^Q = F_k^P + b^{PQ} \ge F_k^P = s_k(P)$$
since as Q extends P, the distance, b^{PQ}, from the reference line in P to that of Q is non-negative.
- If $k \in FU_P \cap FF_Q$ then (from the proof of Lemma 3.1)
$$s_k(Q) = (F_k^Q - b^{PQ}) + (b^{PQ} - w_{R_k}^Q) \ge F_k^Q - b^{PQ} = F_k^P = s_k(P)$$
since $b^{PQ} \ge w_{R_k}^Q$ as word w_{R_k} must be placed after the reference line of P.

The three remaining cases: $k \in UU_P \cap UF_Q$, $k \in UF_P \cap UF_Q$, and $k \in UF_P \cap FF_Q$ are symmetric to the last three cases. $\square$

It follows from correctness of the $A^\star$ algorithm that

THEOREM 4.2. *The $A^\star$ algorithm using penalty function $s(P)$ will always terminate and will return an optimal allowed layout.*

Of course $s(P)$ is not necessarily the best choice of penalty function. In general, the better the estimate of the future penalty, the better the $A^\star$ algorithm will perform. We now give a tighter lower bound, $r(P)$, on the final penalty resulting from partial layout $P = \langle i, j, F, p, h\rangle$.

First consider a figure F_k s.t. $k \in FU_P$, i.e. F_k has been placed but the reference to it, w_{R_k}, has not. One might think that the earliest point that w_{R_k} can occur can be found by extending P to a new layout P' obtained by doing add-line-of-text operations until the word w_{R_k} is placed in a line, and taking $w_{R_k}^{P'}$ to be a lower bound for the placement of w_{R_k} in any layout. Unfortunately, as illustrated in Figure 4, this is not always true, because of the vagaries of text layout.

However, if we abstract away from the details of text layout and treat the text as a fluid then we can find a true lower bound. The idea is to approximate the text by its area. (Similar ideas are used in [3, 4].) We compute the area A of the text $w_{i+1}, ..., w_{R_k}$ (including whitespace between words) and then compute the position of the bottom

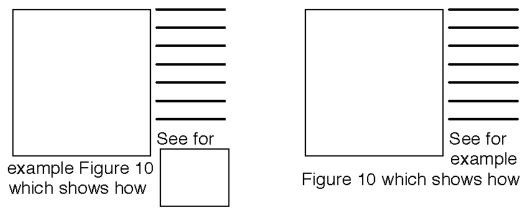

Figure 4: Example showing that adding an extra figure can make a textual reference move earlier.

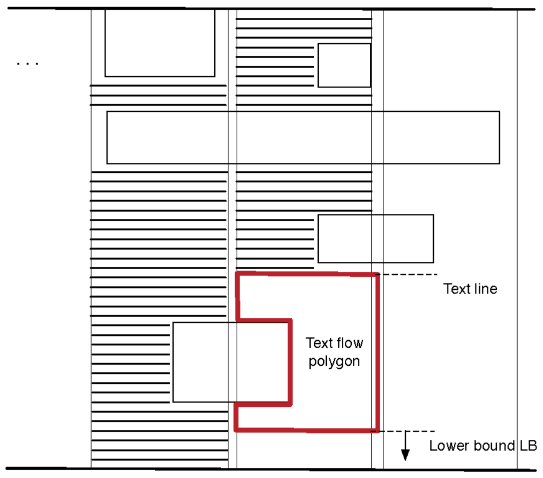

Figure 5: Example showing how approximating text by its area allows the computation of a lower bound on the placement of a dangling text reference.

LB of the text flow polygon whose area is A. The text flow polygon has the text line of P as its top, LB as its bottom, and the sides are the current column plus the width of the widest amount of whitespace at a word break (i.e. an upper bound of how much width is trimmed where line-breaks occur) with cutouts for intruding figures. Figure 5 illustrates the idea. If the bottom of the column is reached then the polygon is extended in the obvious way to the next column. Now LB gives a lower bound on the distance in column inches from the reference line to the bottom of the line containing w_{R_k}. Thus

$$LBW_k^P = \max\{TL^P, LB^P - h_{R_k}\}$$

is a lower bound on the distance in column inches from the reference line to the top of the line containing w_{R_k}, where h_{R_k} is an upper bound on the height of a line containing w_{R_k}, TL^P is the distance from the reference line to the text line, and LB^P is the distance from the reference line to LB.

Now consider a figure F_k s.t. $k \in UF_P$, i.e. F_k has not been placed but the reference to it, w_{R_k}, has been placed. We wish to compute a lower bound on the distance from the reference line to the top of figure F_k. Our layout convention requires that all of figures $F_{j+1}, ..., F_{k-1}$ are placed in order before F_k is placed. Furthermore, there can be at most two figures next to each other in a column.

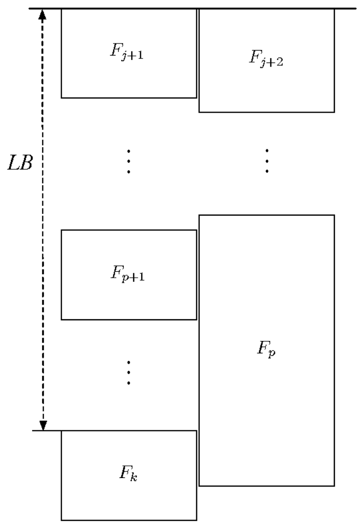

Figure 6: Example showing a two column stacking of figures $F_{j+1}, ..., F_k$ placed in sequential order.

Assume that F_k is placed on one side of the column and that the last figure stacked on the other side of the column is F_p where $j+1 \leq p < k$. This is shown in Figure 6. Note that the top of F_{p+1} must occur no earlier than the top of F_p. The figures F_r, for $r = j+1, ..., p-1$ are at best stacked side by side and so their height must be at least $\frac{\sum_{r=j+1}^{p-1} H_r}{2}$, where H_r is the height of figure F_r. The figures F_r, for $r = p+1, ..., k-1$ are stacked on top of each other so have height $\sum_{r=p+1}^{k-1} H_r$. Thus,

$$LB \geq \frac{\sum_{r=j+1}^{p-1} H_r}{2} + \sum_{r=p+1}^{k-1} H_r$$

where LB is the distance from the reference line to the top of F_k.

A safe lower bound is found by considering all choices for p and choosing the one that leads to the smallest lower bound:

$$LBF_k^P = \min_{p \in \{j+1,...,k-1\}} \frac{\sum_{r=j+1}^{p-1} H_r}{2} + \sum_{r=p+1}^{k-1} H_r.$$

We make this lower bound a bit tighter by taking into account the height of any previously placed figures which protrude below the reference line of P. Assume that figures F_L and F_R protrude below the line with F_L on the left and F_R on the right. Then we introduce two new figures F'_L and F'_R whose height is the amount F_L and F_R, respectively, protrude below the reference line. LBF_k^P is then the lower bound for stacking the figures $F'_L, F'_R, F_{j+1}, ..., F_k$.

Our tighter penalty function $r(P)$ is defined by

$$r(P) = s(P) + \sum_{k \in FU_P} LBW_k^P + \sum_{k \in UF_P} LBF_k^P.$$

Note that if P is a complete layout, then all figures and references are placed, so

$$r(P) = s(P) = p(P).$$

Example	Number of floats	Number of word breaks	Float:text area ratio
Newspaper	9	5634	0.069
Diode	11	5902	0.13
Picasso	18	6231	0.13
Trager	16	13691	0.15
Arch	29	1927	0.39
Yarnivorous	36	5984	0.36
adoptasheep	46	12163	0.42
njhblog	20	5032	1.83

Table 1: Statistics about the example documents used in the empirical evaluation.

It follows from the conservative nature of LBW_k^P and LBF_k^P that:

LEMMA 4.3. *Penalty function $r(P)$ is monotonic.*

THEOREM 4.4. *The $A^\star$ algorithm using penalty function $r(P)$ will always terminate and will return an optimal allowed layout.*

5. EVALUATION

In this section, we compare the automatic float placement algorithms with each other. The main application of our work is for documents that have multiple floats that are linked to the narrative. Such documents include blogs, encyclopedia articles, technical papers and articles. Our eight examples are taken from these different applications. Four are from Wikipedia: the English Wikipedia articles for Arch, Diode, Newspaper, and Pablo Picasso (the last version before 14 March 2007 in each case); a paper on text directionality (named 'Trager' in the tables, after its author); two blogs ('njhblog' and 'Yarnivorous'); and a fairly long publicity piece with lots of pictures ('adoptasheep'). The number of word breaks, floats and the ratio of float area to text area are given in Table 1. The exact texts used can be found at `http://bowman.infotech.monash.edu.au/~pmoulder/scroll-documents.tar.gz`.

All experiments were conducted on a 3.2GHz Pentium 4 with 1GB RAM. All examples were laid out with a column width of 33 body-type ems ($1\frac{1}{2}$ alphabets) and a height of the tallest figure. The allowed placement options for each figure depended upon its width, as follows. We required narrow figures (those no more than $\frac{1}{2}$ of the column width) to be placed on the left or right of a column, and wider figures to be centered on a column or gap. Furthermore, if a column- or gap-centered figure "ate" into a column then this needed to be more than $\frac{1}{4}$ of the column width for this to be a possible placement option. When centering on a column or gap, we only allowed text to flow past the side of the float when the float took up no more than two thirds of each of the two end columns of the float. The result of these placement rules is that all figures have either one or two allowed placements, with most having two. The source code for our implementation can be found at `http://bowman.infotech.monash.edu.au/~pmoulder/scroll-src.tar.gz`.

In our first experiment, we used the dynamic programming algorithm given in Section 3. This was not practical, using 128 seconds of CPU time and more than 1GB of memory for even our smallest example ('Newspaper').

Example	Dynamic Programming Leech	Dynamic Programming Oracle	$A^\star$ $s(P)$	$A^\star$ $r(P)$
Newspaper	1.50s	0.38s	0.46s	0.12s
Diode	3.22s	1.13s	1.43s	0.42s
Picasso	21.79s	4.36s	5.81s	1.61s
Trager	31.95s	10.00s	13.23s	6.45s
Arch	43.51s	12.10s	15.37s	6.46s
Yarnivorous	186.42s	73.68s	99.71s	63.26s
adoptasheep	–	–	–	161.88s
njhblog	–	–	–	–

Table 2: CPU-time comparison between Dynamic Programming (with the single priority queue approach) and $A^\star$ with two different penalty functions: $s(P)$ (the same penalty function used by the dynamic programming implementation) and a tighter lower bound $r(P)$.
Dashes indicate examples where execution was terminated after using over 1GB of memory and two minutes of CPU time.

Example	#Pushes	#Pops	#Expands	Time
Newspaper	48512	42105	36824	0.46s
Diode	127121	105196	100417	1.43s
Picasso	467461	407032	354105	5.81s
Trager	896423	835042	808789	13.23s
Arch	1173729	1099254	927944	15.37s
Yarnivorous	5472243	5271843	4874998	99.71s

Table 3: Results of the $A^\star$ algorithm when using partial layout penalty function $s(P)$, the same as the dynamic programming algorithm.

A standard approach to improve the performance of dynamic programming algorithms is to use an upper bound u on the penalty $p(P)$ of an optimal layout P and discard any partial layout P' s.t. $s(P') > u$ and not place such layouts in the priority queue. (It follows from Lemma 4.1 that if $s(P') > u$ then $p(P) > u$ for all complete layouts P starting from P'.) In our second experiment, we used "leeching" to iteratively find such an upper bound: we start with a small upper bound and see if the dynamic-programming-like algorithm can find a complete layout; if not then we multiply the bound by 1.5 and try again, repeating this until an optimal layout has been found. In Table 2 we give the time taken to determine the layout using this technique ('leech'). We also give the time taken assuming that we have the best possible upper bound, i.e. the penalty for the optimum ('oracle'). This gives a lower bound on how long the dynamic programming algorithm must take when using a penalty bound to prune the search space.

In our third experiment, we measured the time to find the optimal layout using the $A^\star$ algorithm with the two penalty functions $s(P)$ and $r(P)$. When using the same penalty function $s(P)$ as dynamic programming, $A^\star$ is slightly slower than dynamic programming with an exact upper bound provided by an oracle. (We are not sure why; much of the difference remains even after adding the oracle optimization to the $A^\star$ approach and removing the closed-set-clearing optimization from the dynamic programming approach. It could be to do with the more contiguous memory access pattern of dynamic programming.)

Example	#Pushes	#Pops	#Expands	Time
Newspaper	12863	9437	8574	0.12s
Diode	40617	26448	25569	0.42s
Picasso	136014	112676	97479	1.61s
Trager	466845	400954	387415	6.45s
Arch	415178	368308	309481	6.46s
Yarnivorous	3138703	2967382	2735760	63.26s
adoptasheep	7976858	7617804	7043404	161.88s

Table 4: Results of the $A^\star$ algorithm when the tighter lower bound function $r(P)$ for partial layouts.

$A^\star$ using the stronger heuristic $r(P)$ is (as we had hoped) faster than when using the weaker $s(P)$, and is also faster than dynamic programming even with an exact upper bound ('oracle') for these examples. Tables 3 and 4 show further comparisons between using the two penalty functions. '#Pushes' shows how many partial layouts were constructed and added to the priority queue; '#Pops' shows how many of these made it to the front of the priority queue, while '#Expands' shows how many of these were not pruned from the closed set, and formed the starting layout for the new partial layouts pushed onto the priority queue.

While the speed of the $A^\star$ algorithm with penalty function $r(P)$ is adequate for the smaller examples, it does not really scale up to more difficult examples. In particular, the method does not run to completion with the example 'njhblog'. The probable reason is the very high float:text area ratio in this example, which tends to cause more interaction among the figures and a less predictable penalty.

We therefore experimented with techniques for improving the speed of the $A^\star$ algorithm with penalty function $r(P)$. We first tried pruning with an upper bound but even using the best possible upper bound did not change the timings significantly. However, if we do not want a guarantee of optimality we can modify the $A^\star$ algorithm so that it only looks at a sliding "window" of figures. The algorithm is extended by keeping a counter mf which is the maximum figure index for the partial layouts considered so far. Then, whenever a partial layout $P = \langle i, j, F, p, h\rangle$ is removed from the stack for extension, the algorithm checks that $j \geq mf - W$ where $W \geq 0$ is the size of the window. Only if it is in this window is the partial layout expanded. This approach also allows partial layouts before the window in the priority queue to be deleted, thus reducing memory requirements.

In our fourth experiment, we look at the effectiveness of using the sliding window. The results are shown in Table 5. We used a sliding window of sizes from 0 to 4 with $A^\star$ using the $r(P)$ penalty function. We give the time taken to find the best layout as well as a measure of the quality of the layout found. With a window of size 2, all examples except 'njhblog' run in no more than about a second, and the quality of layout is similar to that in the optimal layout except for the 'Arch' example. The example 'njhblog' now runs to completion but still takes almost two minutes of CPU time.

6. EXTENSIONS

The algorithms given here are fairly simplistic in their treatment of the aesthetic criteria. One of the great strengths of the dynamic programming and $A^\star$ algorithms are that they are very general approaches, and can also be used with different (more complex) aesthetic criteria and restrictions on allowed layout.

In fact we have extended the algorithms described here to better take into account aesthetic criteria AC5 and AC6 by restricting figure placement so there is a minimum vertical gap between figures, allowing the insertion of white space at the end of the column, and extending the penalty function to penalize orphan and widow lines and placement of headings at the bottom of a column. This extension was used to generate the layout shown in Figure 1. The main reason for describing the simpler algorithms are that these extensions require more state is kept in a partial layout thus complicating the description and proofs of correctness.

Requiring that float placement not cause a line of text to be on its own requires more state (expressing how much text we have since the previous float or beginning of column), and hence less opportunity for pruning from concluding that two partial layouts have the same set of possible futures, but there's more pruning from rejecting layouts as infeasible (not meeting this layout constraint); the net effect is a moderate speedup in our implementation.

Allowing a column to finish early by a line or two slows the algorithm, partly because of the extra choices generated, and partly because our heuristic penalties become less accurate at predicting which partial layout will lead to the best complete layout.

We have also tried combining optimal float placement with optimal line breaking. We used the conceptually simplest approach (easy to implement), where the set of valid extensions for a partial layout is extended by the possibility of placing fewer words on the line. As one would expect, the extra layout possibilities to search through make the algorithm perform noticeably slower. Our experience with this extension highlighted the importance of properly balancing text penalty against float–reference distance penalty: i.e. deciding to what extent it is acceptable to use word spacing to move references to reduce float–reference distances.

Probably a better approach would be to separate the tasks of float placement and optimal linebreaking, first doing float placement and then holding that placement fixed (or allowing only minimal movement) while improving linebreaking. This would be cheaper, with the time taken being merely the sum of that of the two optimizations rather than involving significant interaction.

We have not implemented a distinction between column breaks and page breaks (as required for print media), but we expect that if page breaks were allowed then the algorithm would perform faster due to less interaction between floats across page breaks (and also there being slightly fewer possible layouts for a document, forbidding floats from spanning across more than one page). Note that, with page breaks, positions in the partial layout must be effectively specified relative to the beginning of the page [assuming that each page has the same arrangement of columns] rather than relative to the reference line (in effect relative to the beginning of the column). This theoretically reduces the opportunity for closed-set pruning, though in practice if two partial layouts have the same next word number and next float number and are up to the same position in their column, then it is almost certain that they would also be in the same column in their page (and same page in their document). This might change if one allowed the input to specify a column break (as distinct from a page break) before headings (for example).

	Window of 0		Window of 1		Window of 2		Window of 3		Window of 4		No window	
Example	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time
Newspaper	2.80	0.02s	1	0.02s	1	0.06s	1	0.10s	1	0.11s	1	0.12s
Diode	1.99	0.02s	1.01	0.04s	1.01	0.11s	1	0.20s	1	0.31s	1	0.42s
Picasso	2.13	0.02s	1.13	0.05s	1	0.16s	1	0.27s	1	0.52s	1	1.61s
Trager	1.43	0.08s	1.42	0.46s	1	0.75s	1	1.37s	1	2.31s	1	6.45s
Arch	1.85	0.02s	1.47	0.02s	1.47	0.06s	1.07	0.06s	1.07	0.14s	1	6.46s
Yarnivorous	1.71	0.03s	1.16	0.15s	1	0.31s	1	0.97s	1	2.51s	1	63.26s
adoptasheep	1.29	0.05s	1.04	0.36s	1	1.24s	1	3.64s	1	7.44s	1	161.88s
njhblog	$\geq$ 1.39	0.06s	$\geq$ 1.09	15.28s	$\geq$ 1.03	154.52s	$\geq$ 1.00	304.13s	$\geq$ 1.00	556.25s	1	$>$ 720s

Table 5: CPU-time and quality of layout for sliding windows of sizes 0 to 4 with $A^\star$ using the $r(P)$ penalty function. 'Cost' is the ratio of the penalty of the best layout found with that window size to the penalty of the optimal layout for that example. Cost entries marked '$\geq$' are where we do not know the optimal solution, so the '$\geq$' numbers are relative to the best solution we have found.

Another interesting extension is allowing alternatives for text and floats. For example, one common approach for a human to improve a layout is to change the text or floats to something very slightly worse in exchange for better layout. From a search perspective, this is simply a matter of extending the set of valid extensions, much like how each float already has multiple possible alignments with corresponding penalties. Most of the programming difficulty is just in the input language and the interface for specifying alternatives and their associated penalties.

7. CONCLUSION

Our research suggests that automatic float placement is necessary if on-line documents are to adapt their appearance to their viewing environment while providing the more sophisticated multi-column design of the kind typically found in magazines. We have identified a number of aesthetic criteria that we think are important in such layout. We have developed algorithms based on dynamic programming and on $A^\star$ for automatic float placement in multicolumn documents that find a layout best satisfying these criteria.

Unfortunately, although these algorithms are fast enough for small examples, they do not scale to larger examples, especially those with many large figures near each other. For this reason, we also investigated an incomplete heuristic modification of the $A^\star$ algorithm in which there was a small window for figure placement. This was considerably faster and seems practical for moderately sized documents. Of course, there is considerable scope for investigating other incomplete approaches to automatic figure placement. This is something we plan to explore further.

8. REFERENCES

[1] A. Brüggemann-Klein, R. Klein, and S. Wohlfeil. Pagination reconsidered. In *Electronic Publishing*, volume 8, pages 139–152, 1995.

[2] J. Felici. *The Complete Manual of Typography.* Peach Pit Press, Berkeley, CA, 2003.

[3] N. Hurst, K. Marriott, and P. Moulder. Toward tighter tables. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*, pages 74–83, New York, NY, USA, 2005. ACM Press.

[4] N. Hurst, K. Marriott, and P. Moulder. Minimum sized text containment shapes. In *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, pages 3–12, New York, NY, USA, 2006. ACM Press.

[5] C. Jacobs, W. Li, E. Schrier, D. Bargeron, and D. Salesin. Adaptive grid-based document layout. *ACM Trans. Graph.*, 22(3):838–847, 2003.

[6] H. W. Lie. CSS3 module: Multi-column layout. Jan. 2001. `http://www.w3.org/TR/2001/WD-css3-multicol-20010118/`.

[7] F. Mittelbach. Formatting documents with floats – a new algorithm for LaTeX 2_ε*. *TUGboat*, 21, 2000.

[8] M. F. Plass. *Optimal pagination techniques for automatic typesetting systems.* PhD thesis, Stanford University, 1981.

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